МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМ.ІГОРЯ СІКОРСЬКОГО»

Кафедра обчислювальної техніки

В.Симоненко, А.Симоненко

СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ
ЛАБОРАТОРНИЙ ПРАКТИКУМ

Навчальний посібник для здобувачів ступеня бакалавр
за освітньою програмою «Комп'ютерні системи та мережі»
спеціальності 123 «Комп'ютерна інженерія»

Електронне мережне навчальне видання

Затверджено
на засіданні кафедри
обчислювальної техніки
ФІОТ НТУУ «КПІ ім. І.Сікорського»
Протокол № 10 від 25.05.2022

Київ
КПІ ім. Ігоря Сікорського
2022

ОС. Лабораторна робота №1

Алокатор пам’яті загального призначення (частина 1)

1. Теоретичні відомості

1.1 Управління пам’яттю

Частина операційної системи, що відповідає за управління пам’яттю, називається модулем управління пам’яттю або менеджером пам’яті (memory allocator). Він спостерігає за тим, яка частина пам’яті використовується в даний момент, а яка – вільна; при необхідності виділяє пам'ять процесам і по їх завершенню звільнює ресурси; управляє обміном даних між оперативною пам’яттю та диском,якщо пам'ять занадто мала для того, щоб вмістити всі процеси.

Прийнято виділяти три основних аспекти управління пам’яттю:

1. **Початкове розподілення.** У момент початку виконання програми кожний блок пам’яті може бути або відведений для деякої цілі, або залишитися вільним. Якщо блок спочатку вільний, то його можна розподілити динамічно у процесі виконання. Для будь-якої системи управління пам'яттю потрібен якийсь метод, що дозволив би системі врахувати вільну пам'ять. Необхідні також механізми виділення вільної пам'яті, якщо в ній виникає необхідність під час виконання.

2. **Утилізація пам'яті.** Пам'ять, яка була розподілена і використовувалася протягом якогось часу, а потім стала непотрібною, повинна бути виявлена системою управління пам'яттю з метою її повторного використання. Утилізація може бути як дуже простою, у випадку переміщення покажчика стека, так і складною, у випадку збору сміття.

3. **Ущільнення та повторне використання.** Після утилізації пам'ять може стати відразу придатною для повторного використання або може знадобитися її ущільнення для побудови великих блоків вільної пам'яті з маленьких.

1.2 Управління пам'яттю за допомогою зв'язного списку

1.2.1 Загальні відомості

Ідея даного методу стає зрозумілою із самої назви: вся оперативна пам'ять представляється менеджером пам'яті у вигляді лінійного зв'язаного списку блоків пам'яті - вільних або зайнятих.

Структура блоку для двохзв'язного списку в загальному випадку має такий вид:

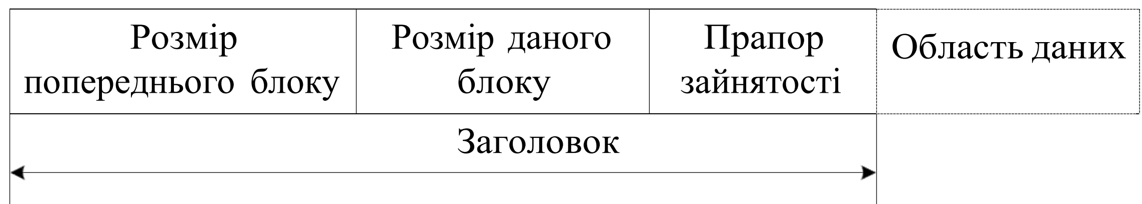


Рис. 1. – Структура загального блоку пам'яті

Поля «Розмір попереднього блоку» і «Розмір даного блоку» зберігають довжину області даних в байтах попереднього і поточного блоків відповідно. Це дозволяє для будь-якого обраного елемента у списку визначити зміщення попереднього і наступного елементів, а також довжину області даних поточного. Розміри даних полів вибираються в залежності від обмежень на максимально можливу довжину блоку для даної системи.

Прапор зайнятості вказує на те, чи вільна дана ділянка пам'яті для використання або вона вже зайнята іншим процесом. Область даних - поле змінної довжини, безпосередньо містить дані процесу.

1.2.2 Виділення пам'яті

Процедура виділення пам'яті складається з двох етапів:

1. Пошук вільного блоку пам'яті необхідного розміру.

2. Поділ знайденого блоку на два незалежних блоки.

Перший етап здійснюється по якомусь алгоритму, який вибирається в залежності від вимог до системи. Більш докладний опис відомих алгоритмів пошуку блоків представлено нижче.

Процедура поділу вільного блоку включає в себе наступні кроки:
1. Зміна заголовка вихідного блоку: установка розміру поточного блоку відповідно до переданих параметрів, установка прапора зайнятості в стан «зайнятий».
2. Створення заголовка другого блоку. Початок заголовка розташовується безпосередньо після області даних попереднього блоку.

Початковий стан пам'яті:

n	m	Не занятий	Область пам'яті, вільної для розподілення

Після успішного запиту на виділення k байт:

n	k	Занятий	k байт	k	d	Не занятий	Область пам'яті, вільної для розподілення

Рис. 2. - Схема виділення пам'яті

n - розмір області даних попереднього блоку;

m - розмір області даних вихідного (вільного) блоку;

d = m - (k + h), де h - довжина блочного заголовку.

Розглянемо приклад. Доступна область пам'яті має розмір 2Кб. Отже, максимальний розмір блоку становить 2Кб, а значить поля довжини блоків будуть складатися з 11 біт. В якості індикатора вільного блоку домовимося використовувати нульовий біт. В такому випадку, розмір заголовка блоку складе 23 біта. Для зручності збільшимо розмір заголовка до 3-х байт - 24 біт. Тоді, повністю вільна пам'ять буде мати наступний вигляд:

0	4096	0	
0 10	11 21	22 23	32767

Рисунок 3. – Початковий блок у пам'яті

Виділимо область пам’яті розміром 16 байт:

0	16	1			4096	0	
0		2	18	19		21	2047

Рис.4. – Стан пам’яті після виділення 16-ти байт

1.2.3 Алгоритм first-fit

Якщо процеси та вільні ділянки зберігаються в списку, відсортованому за адресами, існує кілька алгоритмів для надання пам'яті процесу, створюваному заново (або для існуючих процесів, завантажуваних з диска). Припустимо, менеджер пам'яті знає, скільки пам'яті потрібно надати. Найпростіший алгоритм являє собою вибір першої відповідної ділянки. Менеджер пам'яті переглядає список областей до тих пір, поки не знаходить достатньо великий вільну ділянку. Потім ця ділянка ділиться на дві частини: одна віддається процесу, а інша залишається вільною. Так відбувається завжди, крім статистично нереального випадку точного відповідності вільної ділянки та процесу. Це швидкий алгоритм, тому що пошук зменшено настільки, наскільки можливо.

Однак, при використанні first-fit з лінійним двонаправленим списком виникає специфічна проблема. Якщо кожного разу переглядати список з одного і того ж місця, то великі блоки, розташовані ближче до початку, будуть частіше видалятися. Відповідно, дрібні блоки будуть мати тенденцію накопичуватися на початку списку, що збільшить середній час пошуку. Простий спосіб боротьби з цим явищем полягає в тому, щоб переглядати список то в одному напрямку, то в іншому. Більш радикальний і ще простіший метод полягає в тому, що список робиться кільцевим, і пошук кожен починається з того місця, де ми зупинилися минулого разу. У це ж місце додаються звільнені блоки. У результаті список дуже ефективно перемішується і ніякого «антисортування» не виникає.

1.2.4 Алгоритм next-fit

Алгоритм «наступна придатна ділянка» діє з мінімальними відмінностями від правила «перший придатний». Він працює так само, як і перший алгоритм, але всякий раз, коли знаходить відповідний вільний фрагмент, він запам'ятовує його адресу. І коли алгоритм наступного разу викликається для пошуку, він стартує з того самого місця, де зупинився в минулий раз замість того, щоб кожен раз починати пошук з початку списку, як це робить алгоритм «перший придатний». Моделювання роботи алгоритму, виконане Бейсом, показало, що продуктивність схеми «наступний придатний» трохи гірша, ніж «перший придатний».

1.2.5 Алгоритм best-fit

Інший добре відомий алгоритм називається «самий придатний фрагмент». Він виконує пошук за всім списком і вибирає найменший за розміром відповідний вільний фрагмент. Замість того щоб ділити велику незайняту область, яка може знадобитися пізніше, цей алгоритм намагається знайти фрагмент, близько підходить до дійсно необхідним розмірам. У загальному випадку best-fit збільшує фрагментацію пам'яті. Дійсно, якщо ми знайшли блок з розміром більше заданого, ми повинні відокремити "хвіст" і позначити його як новий вільний блок. Зрозуміло, що в разі best-fit середній розмір цього хвоста буде маленьким, і ми в результаті отримаємо велику кількість дрібних блоків, які неможливо об'єднати, тому що простір між ними зайнято.

Щоб привести приклад роботи алгоритмів «перший придатний» і «самий придатний», знову звернемося до рис. 2. Якщо необхідний блок розміром 2, правило «перший придатний» надасть область за адресою 5, а схема «самий придатний» розмістить процес у вільному фрагменті за адресою 18.

Алгоритм «самий придатний» повільніше «першого придатного», тому що кожного разу він повинен проводити пошук у всьому списку. Але, що трохи дивно, він видає ще більш погані результати, ніж «перший придатний» або «наступний придатний», оскільки прагне заповнити пам'ять дуже маленькими, марними вільними областями, тобто фрагментує пам'ять. Алгоритм «перший придатний» в середньому створює великі вільні ділянки.

У ситуаціях, коли ми розміщуємо блоки декількох фіксованих розмірів, алгоритми best-fit виявляються краще. Проте бібліотеки розподілу пам'яті розраховують на гірший випадок, і в них зазвичай використовуються алгоритми first-fit.

1.2.6 Алгоритм близнюків

У разі роботи з блоками декількох фіксованих розмірів напрошується таке рішення: створити для кожного типорозміру свій список. Це позбавляє програміста від необхідності вибирати між first- та best-fit, усуває пошук в списках як явище.

Цікавий варіант цього підходу для випадку, коли різні розміри є ступенями числа 2, як 512 байт, 1 Кбайт, 2Кбайта і т.д., називається алгоритмом близнят. Він полягає в тому, що ми шукаємо блок необхідного розміру у відповідному списку. Якщо цей список порожній, ми беремо список блоків вдвічі більшого розміру. Отримавши блок удвічі більшого розміру, ми ділимо його навпіл. Непотрібну половину ми поміщаємо у відповідний список вільних блоків. Цікаво, що нам абсолютно неважливо, чи отримали ми цей блок просто з відповідного списку, або ж діленням навпіл вчетверо більшого блоку, і далі по рекурсії. Одна з переваг цього методу полягає в простоті об'єднання блоків при їх звільненні. Дійсно, адреса блоку-близнюка виходить простим інвертуванням відповідного біта в адресі нашого блоку. Потрібно тільки перевірити, чи вільний цей близнюк. Якщо він вільний, то ми об'єднуємо братів в блок удвічі більшого розміру, і т.д.

Алгоритм близнюків значно знижує фрагментацію пам'яті і різко прискорює пошук блоків. Найбільш важливою перевагою цього підходу є те,що навіть у найгіршому випадку час пошуку не перевищує $O(\log(S_{max})-\log(S_{min}))$, де S_{min} і S_{max} позначають відповідно мінімальний і максимальний розміри використовуваних блоків. Це робить алгоритм близнят важко замінним для ситуацій, коли необхідний гарантований час реакції - наприклад, для задач реального часу. Часто цей алгоритм або його варіанти використовуються для виділення пам'яті усередині ядра ОС. Наприклад,

функція kmalloc, використовувана в ядрі ОС Linux, заснована саме на алгоритмі близнюків.

1.2.7 Алгоритм worst-fit

Намагаючись вирішити проблему поділу пам'яті на області та маленькі вільні фрагменти,які практично точно збігаються з процесом, можна замислитися про алгоритм «сама невідповідна ділянка». Він завжди вибирає найбільшу вільну ділянку, від якого після поділу залишається область достатнього розміру і її можна використовувати в подальшому. Однак моделювання показало, що це також не дуже хороша ідея.

Всі чотири алгоритму можна прискорити, якщо підтримувати окремі списки для процесів і вільних областей. Тоді пошук проводитиметься тільки серед незайнятих фрагментів. Неминуча ціна, яку потрібно заплатити за збільшення швидкості при розміщенні процесу в пам'яті, полягає в додатковій складності і уповільненні при звільненні областей пам'яті, так як фрагмент, що став вільним, необхідно видалити зі списку процесів і вставити в список незайнятих ділянок.

Якщо для процесів і вільних фрагментів підтримуються окремі списки, то останній можна відсортувати за розміром, тоді алгоритм «самий придатний» працюватиме швидше. Коли він виконує пошук у списку вільних фрагментів від самого маленького до найбільшого, то, як тільки знаходить придатну незайняту область, алгоритм вже знає, що вона найменша з тих, в яких може поміститися завдання, то є найкраща. На відміну від схеми з одним списком, подальший пошук не потрібно. Таким чином, якщо список вільних фрагментів відсортований за розміром, схеми «перший придатний» і «найкращий» однаково швидкі, а алгоритм «наступний придатний» не має сенсу.

За підтримки окремих списків для процесів і вільних фрагментів можлива невелика оптимізація. Замість створення окремого набору структур даних для списку вільних ділянок, як це зроблено на рис. 2, в, можна використовувати самі вільні області. Перше слово кожного незайнятого

фрагмента може містити розмір фрагмента, а друге слово може вказувати на наступний запис. Вузли списку на рис. 2, в, для яких потрібні три слова і один біт (Р / Н), більше не потрібні.

1.2.8 Алгоритм fast-fit

Ще один алгоритм розподілу називається «швидкий придатний», він підтримує окремі списки для деяких з найбільш часто запитуваних розмірів. Наприклад, могла б існувати таблиця з n записами, в якій перший запис вказує на початок списку вільних фрагментів розміром 4 Кбайт, другий запис є покажчиком на список незайнятих областей розміром 8 Кбайт, третій - 12 Кбайт і т. д. Вільний фрагмент розміром , скажімо, 21 байт, міг би розташовуватися або в списку областей 20 Кбайт або в спеціальному списку ділянок додаткових розмірів. При використанні правила «швидкий придатний» пошук фрагмента необхідного розміру відбувається надзвичайно швидко. Але цей алгоритм має той же самий недолік, що і всі схеми, які сортують вільні області за розміром, а саме: якщо процес завершується завантажується на диск, пошук його сусідів з метою дізнатися, чи можливо їх з'єднання, є дорогою операцією. А якщо не робити злиття областей, пам'ять дуже швидко виявиться розбитою на величезне число маленьких вільних фрагментів, в які не поміститься жоден процес.

1.2.9 Звільнення пам'яті

Процедура звільнення блоків пам'яті вимагає об'єднання сусідніх вільних блоків. Можливі 4 випадки:

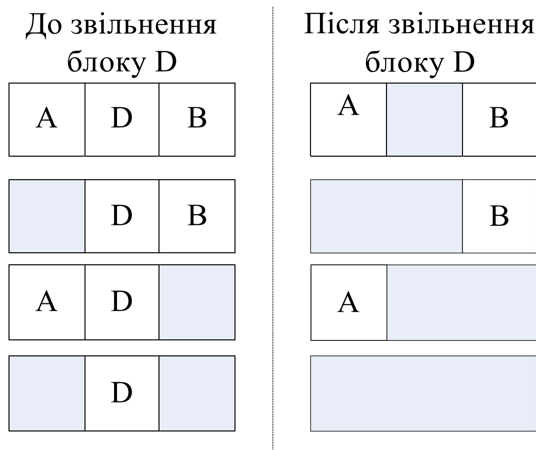


Рис. 5. - Схема звільнення блоку пам'яті

При цьому операції з заголовками блоків виконуються в послідовності, зворотній тій, що зображена на рис. 2.

1.3 Управління пам'яттю за допомогою роздільних списків

1.3.1 Загальні відомості

У методі зв'язаного списку значні ресурси витрачаються на поділ і об'єднання блоків пам'яті. В якості вирішення цієї проблеми можна розглянути метод роздільних списків, який пропонує розділити всю пам'ять на кілька списків, кожен з яких буде пов'язувати блоки деякої фіксованої довжини. Це дозволить переглядати не всі блоки купи, а лише елементи 1 списку.

Крім того, заголовок кожного блоку буде мати тільки 1 поле - прапор зайнятості. Інші поля стануть марними, так як розмір блоку став фіксованим. Розглянемо структуру пам'яті, в якій є 8 розмірів блоків (в байтах): 16, 32, 64, 256, 1024, 2048.

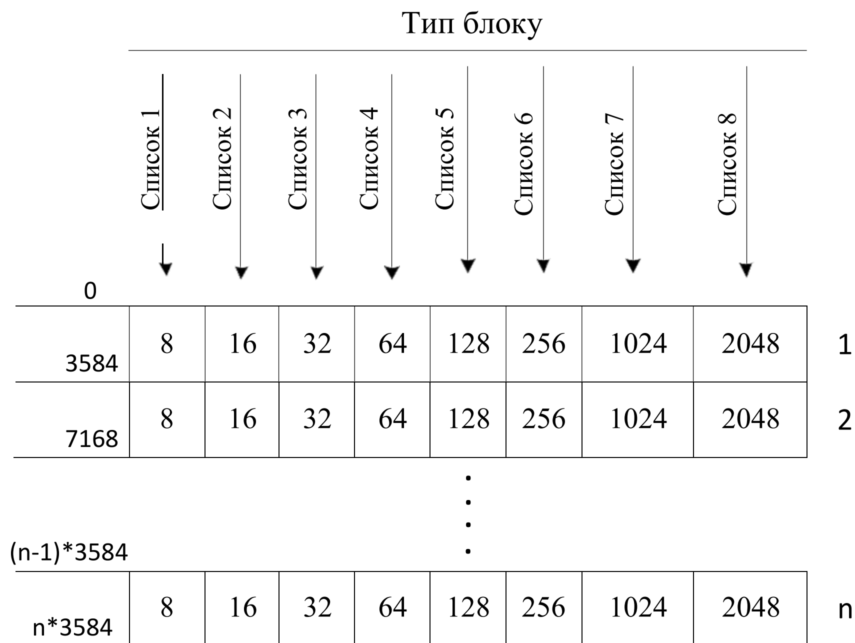


Рис. 6. – Організація пам’яті при використанні розділених списків.

Очевидно, що для перегляду списку певних блоків потрібно до деякого початкового зсуву (адреси першого елемента списку) додавати константне значення (довжину рядка). Так, для елементів 2-го списку процедура перегляду буде виглядати наступним чином:

$a_i = (8 + 1) + 3584 \cdot i$

Слід звернути увагу на додавання 1 при обчисленні початкового зсуву: це розмір заголовка блоку.

1.3.2 Виділення блоку пам'яті

Операція виділення при використанні даної моделі управління пам'яттю складається з трьох етапів:

1. Округлення запитуваного розміру пам'яті A до найближчого константного значення C, причому A ≤ C.

2. Пошук в списку блоків типу C першого вільного блоку. Якщо не знайдений, то функція повертає NULL.

3. Установка прапора зайнятості знайденого блоку в положення «зайнято». Функція повертає значення адреси початку області даних блоку.

Слід зазначити, що пункт № 2 можна реалізувати дещо інакше: в разі, якщо неможливо знайти вільний блок розміром С, то починається перегляд списку блоків наступного типу (розміру). Таке рішення, з одного боку, може привести до підвищення внутрішньої фрагментації пам'яті. Однак, з іншого боку, зробить систему більш надійною і гнучкою.

1.3.3 Звільнення блоку пам'яті

При використанні методу розділених списків операція звільнення пам'яті є безпечною, так як завжди можна з легкістю перевірити коректність адреси, переданої для звільнення. У випадку, якщо покажчик посилається на зайнятий блок, то прапор зайнятості позначається як вільний і процедура успішно завершується. В іншому випадку нічого не відбувається (або виводиться на консоль повідомлення про помилку).

Розглянемо приклад. Припустимо, для моделі пам'яті, представленої на рис. 6, користувач викликав функцію mem_free (((void *) 13762)).

Визначимо, до якого блоку відноситься адреса 13762: $Offs = A \bmod L_{рядка} = 13762 \bmod 3584 = 3010$

$Row = A \setminus L_{рядка} - 1 = 13762 \setminus 3584 - 1 = 2$

Очевидно, що це є блоком типу 2048, 2-й елемент списку. Знайдемо адреса заголовка для цього блоку:

$A_{head}=(8+l)+(l6+l)+(32+l)+(64+1)+(128+l)+(256+l)+(512+l)+(l024+l)+3584\cdot 2= =9216$

Перевіряємо байт за адресою 9216. Якщо він дорівнює 1, то встановлюємо його в 0. Інакше - повідомляємо користувачеві про помилку.

1.3.4 Удосконалення методу (зв'язні списки)

Поєднавши всі блоки одного типу (розміру) в зв'язаний список, можна прискорити пошук вільного блоку заданого розміру. Для цього перед використанням пам'ять слід привести до наступної структури:

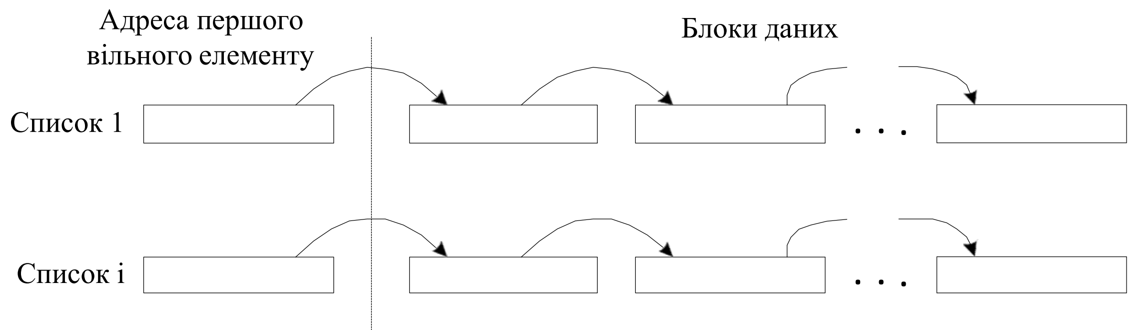


Рис. 7. – Удосконалення методу

З діаграми видно, що були введені додаткові комірки, що містять покажчики на адреси першого в зв'язаному списку вільного заданого типу (розміру). Поки блок не зайнятий, в його молодших адресах зберігається покажчик на наступний незайнятий блок. Коли ж блок розподіляється під деякі дані (а це завжди 1-й елемент у зв'язаному списку), то значення покажчика, що зберігається в ньому, зберігається у відповідній клітинці адреси першого елемента в списку.

Також слід зазначити, що необхідність в заголовку блоку відпала. Єдине застосування, яке міг би знайти прапор зайнятості для блоку це коректність операції звільнення ділянки пам'яті.

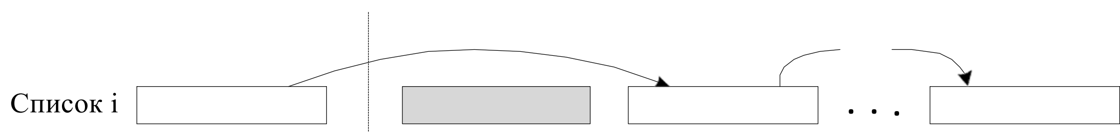


Рис. 8. – Виділення блоку пам'яті у вдосконаленому методі

Звільнення блоку пам'яті виконується в 2 кроки:

1. Збереження в молодших адресах блоку адреси першого вільного елементу списку

2. Запис на адресу першого вільного елементу списку адреси звільнився блоку.

2. Завдання на роботу

2.1 Розробити

Розробити алокатор загального призначення, який реалізує 3 функції, що вирішують завдання менеджера пам'яті. А саме: виділення, звільнення та ущільнення пам'яті. Нижче приведено більш докладний опис необхідного інтерфейсу:

• void * mem_alloc (size_t size) - функція повинна виділити блок пам'яті заданого розміру в size байт. Якщо блок пам'яті був виділений успішно, то повернути адресу початку цього блоку, в іншому випадку повернути NULL.

• void * mem_realloc (void * addr, size_t size) - функція повинна змінити розмір блоку пам'яті з адресою addr до size байт. При цьому вміст (всі або частина) старого блоку пам'яті може бути перенесено в інший блок пам'яті. Якщо вдалося змінити розмір блоку пам'яті, то функція повинна повернути адресу нового блоку пам'яті, інакше повернути NULL, і не зруйнувати старий блок пам'яті. Якщо addr дорівнює NULL, то виклик функції аналогічний виклику mem_alloc (size).

• void mem_free (void * addr) - функція повинна звільнити перш виділений блок пам'яті.

Кілька зауважень до принципів роботи вище описаних функцій:

1. Функцію mem_realloc () можна використовувати як для зменшення, так і для збільшення розміру блоку. Ця функція може виділити новий блок пам'яті за новою адресою, при цьому вміст старого блоку завжди копіюється у новий блок (все в разі збільшення або не зміни розміру блоку, або частину в разі зменшення розміру). Якщо новий блок успішно виділений, го блок пам'яті за старою адресою вважається недійсним. Якщо блок нового розміру виділити не вдалося, то повертається NULL, при цьому старий блок пам'яті повинен бути доступний для використання.

2. Функція mem_free () звільняє виділений блок пам'яті, після виклику цієї функції цей блок вважається недійсним і не може використовуватися програмою.

3. Якщо пам'ять виділена функціями mem_alloc () або mem_realloc (), то її не можна переносити куди-небудь до виклику mem_free () або mem_realloc (), якщо mem_realloc () повернула іншу адресу.

Вимоги до розроблюваної програми:

1. Області пам'яті можна виділяти будь-яким доступним способом.

2. Функції mem_alloc (), mem_realloc () і mem_free () повинні відповідати наведеним вище прототипам.

3. Адреси пам'яті, що повертаються функціями mem_alloc () і mem_realloc (), повинні бути вирівняні на границю в 4 байта.

4. Спробувати зменшити час пошуку вільного блоку пам'яті і час звільнення зайнятого блоку.

5. Спробувати зменшити фрагментацію пам'яті.

6. Написати функцію mem_dump (), яка повинна виводити на консоль стан областей пам'яті.

2.2 Надати звіт

Звіт повинен містити:

1. Опис розробленого алгоритму.

2. Оцінку часу пошуку вільного блоку пам'яті, оцінку часу звільнення зайнятого блоку.

3. Оцінку витрати пам'яті для зберігання службової інформації.

4. Опис переваг та недоліків розробленого алокатора.

5. Лістинг алокатора пам'яті загального призначення.

6. Приклад роботи алокатора.

- Електронна версія (мережа КПІ)

ftp://vt513.comsys.ntu-kpi.kiev.ua/pub/edu/spol/labl-l .doc

- **Література**

1. Bill Blunden. Memory Management: Algorithms and Implementation in

C/C++.

2.Эндрю Таненбаум. Современные операционные системы, 2-е издание.

3. Роберт Седжвик. Фундаментальные алгоритмы на С.

ОС. Лабораторна робота №2

Алокатор пам’яті загального призначення (часть 2)

Коротка теорія

У лабораторній роботі № 1 було розглянуто простий метод побудови алокатора загального призначення. Розглянемо інший метод вирішення цієї ж задачі. Далі описано один з варіантів реалізації.

Вся віртуальна пам'ять складається зі сторінок. Кожна сторінка може знаходитися в оперативній пам'яті або в зовнішньому файлі (розглядаються сторінки, до яких було хоча б одне звернення, тобто сторінки для яких були виділені ресурси віртуальної пам'яті). При зверненні до сторінки, яка знаходиться в зовнішньому файлі, відбувається сторінковий промах і операційна система знаходить вільну фізичну сторінку і зчитує в неї вміст із зовнішнього файлу. Розмір сторінки звичайно дорівнює від 4 Кбайт до декількох Мбайт. Таким чином, якщо в процесі прийняття рішення алокатор пам'яті звертається до меншого числа сторінок, тим менше він залишає слід у пам'яті, тим він ефективніший.

Як і в лабораторній роботі № 1 алокатор пам'яті запитує деяку область пам'яті в операційної системи. Далі вся ця пам'ять ділиться на сторінки. Розмір сторінки не обов'язково повинен збігатися з розміром віртуальної сторінки. Наприклад, одна віртуальна може вміщати декілька сторінок алокатору. Всі сторінки вирівняні, таким чином при зверненні до даних за

будь-якою адресою всередині сторінки ми звертаємося тільки до однієї віртуальної сторінки.

Всі блоки пам'яті діляться на дві групи. У першу групу входять блоки з розмірами менше або рівними половині сторінки, у другу всі інші. Блоки першої групи діляться на класи. Блоки одного класу мають однаковий розмір. Наприклад, цей розмір може бути числом 2x ($x \geq 4$). Якщо додаток запрошує блок пам'яті деякого розміру меншого або рівного половині сторінки, то алокатор призначає цього блоку найближчий за розміром клас. Блоки другої групи це блоки розміром в одну або кілька сторінок. Якщо програма запитує блок пам'яті деякого розміру більшого ніж половина сторінки, то алокатор округлює цей розмір до найближчого цілого числа сторінок.

Кожна сторінка може знаходитися в одному з трьох станів: сторінка вільна, сторінка розділена на блоки одного класу або сторінка зайнята багатосторінковим блоком. Алокатор містить список вільних сторінок. Цей список можна створити відразу після отримання області пам'яті або в міру звільнення зайнятих сторінок. Якщо сторінка розділена на блоки одного класу, то в цій сторінці можуть бути тільки блоки цього класу і всі блоки мають однаковий розмір. Якщо сторінка зайнята багатосторінковим блоком, то після цієї сторінки можуть перебувати нуль або декілька сторінок одного блоку пам'яті.

Для кожної сторінки існує описувач сторінки, який однозначно визначає її стан. Місце знаходження описувача можна визначити по вказівнику на описувач. Всі вказівники на Описувачі сторінок можна розмістити в одномірному масиві (довжина масиву дорівнює числу сторінок у виділеній області пам'яті). Для сторінки розділеної на невеликі блоки описувач сторінки можна зберігати в самій сторінці. Для сторінки розділеної на великі блоки описувач сторінки можна зберігати в блоці меншого розміру (щоб отримати цей блок, необхідно рекурсивно викликати алокатор пам'яті). Для багатосторінкових блоків Описувачі сторінок не потрібні, всю інформацію

про кількість сторінок можна закодувати в покажчиках на Описувачі сторінок багатосторінкового блоку.

Для сторінки розділеної на блоки в описувачі сторінки є вказівник на перший вільний блок в сторінці, а також лічильник кількості вільних блоків в цій сторінці. Всі вільні блоки в одній сторінці пов'язані в список, поля для зв'язування в список знаходяться в самих вільних блоках. При звільненні деякого блоку алокатор за адресою блоку вираховує номер його сторінки, за номером сторінки визначає вказівник на описувач сторінки, додає звільнений блок в список вільних блоків і збільшує лічильник вільних блоків. Якщо всі блоки вільні, то ця сторінка звільняється.

Описувачі сторінок, розділених на блоки одного і того ж класу та у яких є хоча б один вільний блок, пов'язані в список. Сторінки, в яких немає вільних блоків, до цього списку не входять, але якщо у сторінки з'являється один вільний блок, то вона додається до цього списку. В алокаторі пам'яті є масив вказівників на ці списки для кожного класу блоків. При запиті на виділення пам'яті перевіряється список описувачів сторінок потрібного класу. Якщо цей список порожній, то береться порожня сторінка, поділяється на блоки і додається в список. Якщо список не порожній, то береться перша в списку сторінка і в ній перший вільний блок віддається програмі.

Облік вільної пам'яті проводиться за допомогою дерева, де ключем є розмір вільного простору. Пам'ять для вершин дерева береться в самих вільних сторінках, що становлять вільні ділянки пам'яті.

У наведеному вище алгоритмі масив вказівників на Описувачі сторінок не обов’язковий. Для сторінок розділених на невеликі блоки описувач сторінки можна розміщувати прямо в сторінці (ознакою того, що описувач сторінки знаходиться на початку сторінки, може бути адреса блоку). Для всіх інших сторінок Описувачі сторінок можна розміщувати в хеш таблиці, де ключем пошуку є адреса сторінки, пам'ять для хеш таблиці можна отримати за допомогою рекурсивного виклику алокатора.

Завдання на роботу

Розробити алокатор загального призначення, який реалізує 3 функції, що вирішують завдання менеджера пам'яті. А саме: виділення, звільнення та ущільнення пам'яті. Нижче приведено більш докладний опис необхідного інтерфейсу:

• void * mem_alloc (size_t size) - функція повинна виділити блок пам'яті заданого розміру в size байт. Якщо блок пам'яті був виділений успішно, то повернути адресу початку цього блоку, в іншому випадку повернути NULL.

• void * mem_realloc (void * addr, size_t size) - функція повинна змінити розмір блоку пам'яті з адресою addr до size байт. При цьому вміст (всі або частина) старого блоку пам'яті може бути перенесено в інший блок пам'яті. Якщо вдалося змінити розмір блоку пам'яті, то функція повинна повернути адресу нового блоку пам'яті, інакше повернути NULL, і не зруйнувати старий блок пам'яті. Якщо addr дорівнює NULL, то виклик функції аналогічний виклику mem_alloc (size).

• void mem_free (void * addr) - функція повинна звільнити перш виділений блок пам'яті.

Звіт

Звіт повинен містити:

1. Опис розробленого алгоритму.
2. Оцінку часу пошуку вільного блоку пам'яті, оцінку часу звільнення зайнятого блоку.
3. Оцінку витрати пам'яті для зберігання службової інформації.
4. Опис переваг та недоліків розробленого алокатора.
5. Лістинг алокатора пам'яті загального призначення.
6. Приклад роботи алокатора.

Електронна версія (мережа КПІ)

ftp://vt513.comsys.ntu-kpi.kiev.ua/pub/edu/spo1/lab1-2.doc

Література

1. Jeff Bonwick. *The Slab Allocator: An Object-Caching Kernel Memory Allocator*.
2. Uresh Vahalia. *Unix Internals: The New Frontiers.* (є переклад на російську)

ОС. Лабораторна робота №3

Дослідження дисциплін обслуговування заявок при обмежених ресурсах

Ефективність роботи обчислювальної системи залежить не тільки від власної ефективності алгоритмів обробки інформації і технічних характеристик обчислювальної системи, але і від прийнятих в системі правил виконання робіт, прийому і обробки запитів користувачів.

Ефективність методів обслуговування визначається можливістю затримки або втрати заявки до обробки, а також часом знаходження заявки в системі. Залежно від типу системи управління і диспетчеризації, затримка заявок може враховуватися по загальному середньому часу затримки або по допустимому часу очікування.

Під час вивчення дисциплін обслуговування заявок передбачається, що процеси введення і обслуговування є незалежними. Заявка, яка поступає в систему, починає обслуговуватися негайно, якщо у цей момент ресурс для її обслуговування вільний.

Якщо ресурс зайнятий обслуговуванням попередніх заявок, тоді залежно від типу заявки, яка поступила, і прийнятого в системі правила (дисципліни) обслуговування, заявка, що тільки що поступила, може чекати свою чергу або перервати заявку, яка виконується. У разі переривання заявки

передбачається, що вона повертається в чергу, де вона буде чекати продовження перерваного обслуговування. Тривалість перебування кожної заявки у обчислювальній системі складається з часу очікування заявки і часу обслуговування машиною.

Розподіл заявок між ресурсами, які їх виконують і котрі є в наявності носить назву ***планування***. Одним із методів планування, орієнтованих на захоплення ресурсу, є метод черг. Нові заявки знаходяться у вхідній черзі, що часто зветься чергою робіт - завдань (***job queue***) та очікують звільнення ресурсу.

Дисципліна обслуговування - це спосіб визначення того, яка вимога в черзі слід обслуговувати наступним. Рішення може грунтуватися на одній з наведених нижче характеристик або на їх сукупності:

1) міра, обумовлена відносним часом надходження розглянутого вимоги у чергу;

2) міра необхідного або отриманого до цих пір часу обслуговування;

3) функція, що визначає приналежність вимоги до тієї або іншої групи.

Прикладами дисциплін обслуговування є постійно використовувана модель «перший прийшов - перший обслужений» (FCFS-first came-first served), звана в російськомовній літературі «дисципліна обслуговування в порядку надходження»-ОПП.

Основні типи дисциплін обслуговування представлені на рис. 1

Розрізняють два типи дисциплін обслуговування - *без пріоритетні* та *пріоритетні*.

У разі дисциплін без пріоритетів заявки різних типів не мають наперед встановлених пріоритетів для обслуговування і вважаються при вході в систему рівно пріоритетними. При реалізації *пріоритетних* дисциплін обслуговування окремим задачам надається привілейоване право переходу в стан виконання. Пріоритет присвоєний задачі, може бути величиною постійною, або може змінюватись в процесі її розв'язання. У деяких системах вводяться класи пріоритетів (частково впорядковані системи пріоритетів). Це

як правило робиться за рахунок організації декількох черг для кожного класу. Ресурс буде представлений в першу чергу тим заявкам, котрі знаходяться в черзі (класі) з найбільш високим пріоритетом.

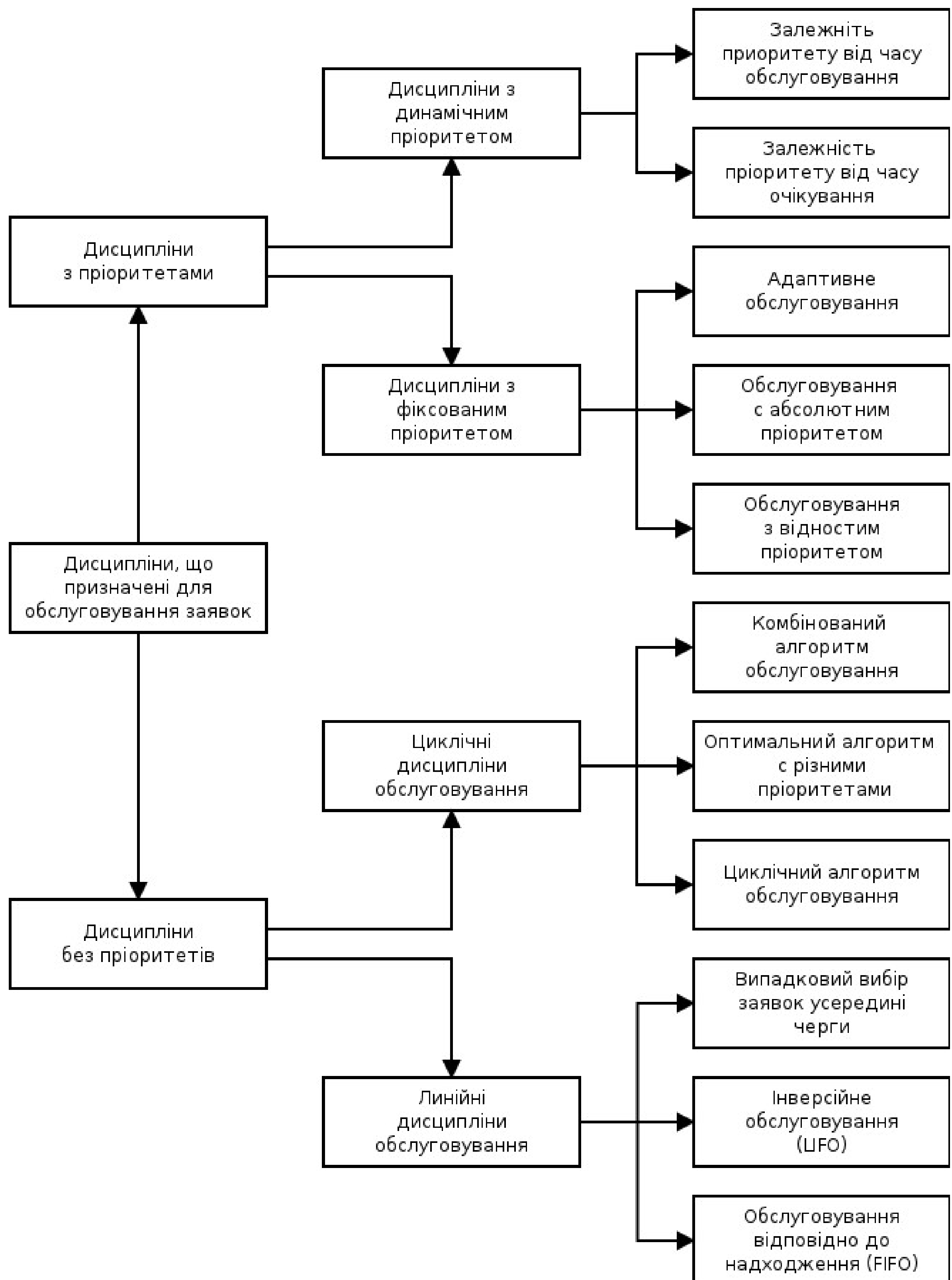


Рис 1.

Витісняючи та не витісняючи алгоритми диспетчеризації.

Диспетчеризація без перерозподілу процесорного часу в час виконування заявки це є ***не витісняюча багатозадачність*** - це такий спосіб диспетчеризації при якому активний процес виконується до тих пір поки він сам не віддасть управління диспетчеру задач для вибору із черги іншого готового до виконання процесу. При не витісняючий багатозначності механізм розподілу процесорного часу розподілений між системою та прикладними програмами. Дисципліна SJB (**Shortest Job First**) відноситься до не витісняючих.

Диспетчеризація з перерозподілом процесорного часу між задачами є ***витісняючою багатозадачністю***. Це такий спосіб при якому рішення про переключення процесору з виконання одного процесу на виконання іншого приймається диспетчером задач, а не активною задачею. При витісняючий багато задачності механізм диспетчеризації задач цілком зосереджений в операційній системі і програміст може писати своє програмне забезпечення не турбуючись про те, як воно буде виконуватись разом з іншими задачами. При цьому ОС виконує наступні функції: визначає момент зняття з виконання поточної задачі, зберігає її контекст; вибирає з черги готових задач наступну і запускає її на виконання наперед завантаживши її контекст. Дисципліну RR та інші дисципліни обслуговування побудовані на її основі відносять до витісняючих.

<u>КРИТЕРІЇ ПЛАНУВАННЯ.</u>

Для порівняння алгоритмів планування використовуються наступні критерії:

1. **використання CPU** (Central Processing Unit utilization). Використання CPU теоретично може знаходитися в межах від 0 до 100%. В реальних системах використання CPU коливається в межах 40% для легко завантаженого CPU, 90% для важко завантаженого CPU.

2. **пропускна здатність CPU** (throughput). Пропускна здатність CPU може вимірюватися кількістю заявок, котрі

виконуються за одиницю часу.

3 **час повернення** (turnaround time) для деяких процесів важливим критерієм є повний час виконання, тобто існує інтервал від моменту появи процесу у вхідній черзі до моменту його завершення. Цей час зветься часом повернення і включає час очікування у вхідній черзі, час очікування в черзі готових процесів, час очікування в чергах до апаратури, час виконання в процесорі та час вводу-виводу.

4 **час очікування** (waiting time). Під часом очікування мається на увазі сумарний час знаходження заявки в черзі до ресурсу.

ДИСЦИПЛІНИ ОБСЛУГОВУВАННЯ ЗАЯВОК.

1. Лінійні дисципліни обслуговування

У випадку дисциплін без приоритетів заявки різних типів не мають супроводжуючих пріоритетів для обслуговування та вважається, що вони мають рівний пріоритет при вході у систему. Ці правила дотримуються тоді, коли заявки для обслуговування обираються відповідно до:

- порядку надходження (першою в отриманні обслуговування буде та заявка, яка прийшла першою, FIFO — First Input First Output, рис 3.)
- порядку, інверсному порядку надходження (першою отримує обслуговування заявка, яка прийшла останньою, LIFO — Last Input First Output, рис 4.)
- заявки для обслуговування обираються з черги випадково.

Ці три дисципліни без пріоритетів характеризуються однаковим часом очікування для всіх заявок. Але дисципліна FIFO мінімізує дисперсію очікування, тому ця дисципліна є більш кращою та переважною серед усіх інших дисциплін.

Алгоритм FIFO

Алгоритм обслуговування черг Firstin, Firstout (FIFO), також званий First Come First Served є найпростішим.

FIFO є найбільш простою стратегією планування процесів і полягає в тому, що ресурс передається тому процесу, котрий раніше всіх інших звернувся до нього. Коли процес потрапляє в чергу готових процесів, process control block приєднується до хвоста черги. Середній час очікування для стратегії FIFO є часто досить великим і залежить від порядку надходження процесів в чергу готових процесів.

Приклад № 1

Нехай три процеси потрапляють в чергу одночасно в момент 0 и мають наступні значення часу послідовного обслуговування в CPU.**варіант 1:**

П1(24 мс)

П2(3 мс)

П3(3 мс)

варіант 2:

П2(3 мс)

П3(3 мс)

П1(24 мс)

На малюнку наведені діаграми Ганта виконування черги готових процесів (WT_n- час очікування)

варіант 1:

$П_1$	$П_2$	$П_3$	Середній час очікування
WT_1=0 мс	WT_2=24 мс	WT_3=27 мс	WT=17 мс

варіант 2:

$П_2$	$П_3$	$П_1$	Середній час очікування
WT_2=0 мс	WT_3=3 мс	WT_1=6 мс	WT=3 мс

Стратегії FIFO притаманний так званий “ефект конвою”. В тому випадку, коли в комп'ютері є один великий процес та декілька малих, то всі процеси збираються на початку черги готових процесів, а згодом в черзі до обладнання. Таким чином, “ефект конвою” призводить до зниження пропускної здатності як процесору, так і периферійного пристрою.

Дисципліна обслуговування FIFO з пріоритетами без витіснення припускає, що кожна заявка має свій пріоритет. Заявки з однаковими пріоритетами групуються в чергу типу FIFO. Спочатку обслуговується черга з вищим пріоритетом. Заявка, що потрапила в процесор не може бути витіснена з нього поки не завершиться її обслуговування.

Алгоритм LIFO є обслуговуванням в зворотному напрямку.

<u>Особливості організації алгоритму FIFO:</u> Реалізація стратегії, за якої завдання закінчуються в порядку надходження. Дві черги готових процесів: одна - для процесів, які вже виконувалися, інша - для тих, хто прийшов.

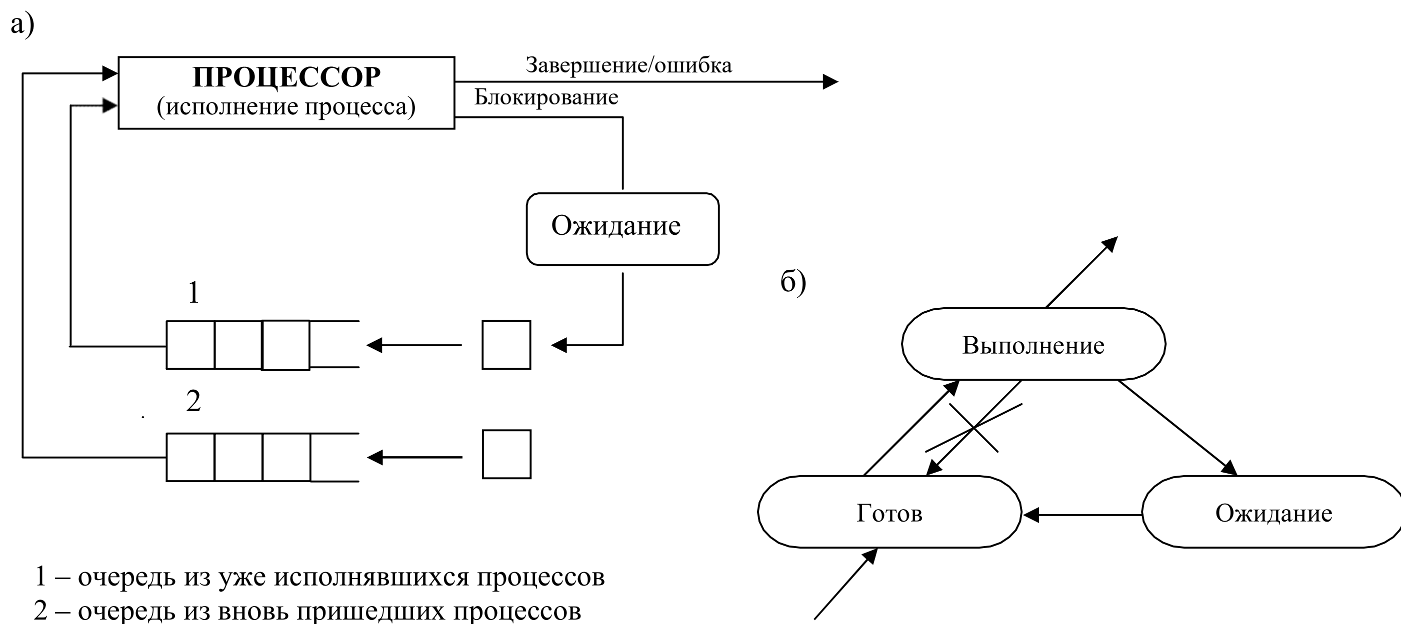


Рис.2. а) Схема линейной дисциплины обслуживания;

б) Граф состояний процесса в системе с линейной дисциплиной обслуживания

2. Циклічні дисципліни обслуговування

З циклічних дисциплін обслуговування у системах з розподіленим часом найбільш практичне застосування мають:

- циклічний алгоритм обслуговування однією чергою (RR - Round Robin, рис 5.);
- багатопріоритетний циклічний алгоритм обслуговування (Fbn - Foregraund-Backgraund, рис. 7);
- багатопріоритетний циклічний алгоритм обслуговування — алгоритм Корбато (рис. 8);
- змішаний алгоритм.

Циклічний алгоритм обслуговування

У випадку циклічного алгоритму планування (RR) заявки отримують обслуговування центральним процесором згідно їх надходженню та протягом визначеного кванту часу. Якщо обслуговування завершується протягом цього кванту, тоді заявка, що отримала обслуговування, покидає ресурс і на обслуговування надходить заявка, яка слідує за нею в черзі; у протилежному випадку заявка, яка не була до кінця обслуговувана, розміщується у кінці черги і повинна чекати наступний квант часу для отримання остаточного обслуговування.

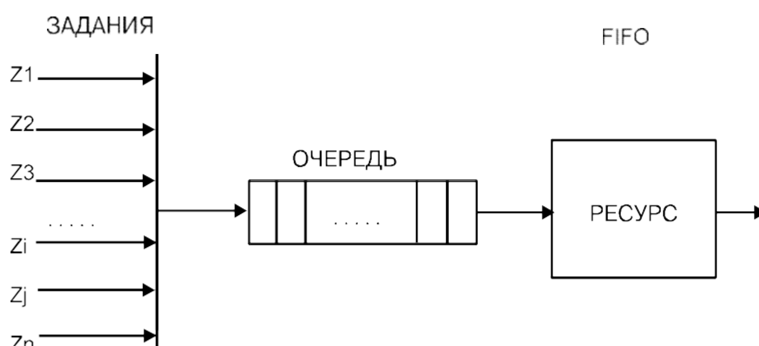


Рис. 3

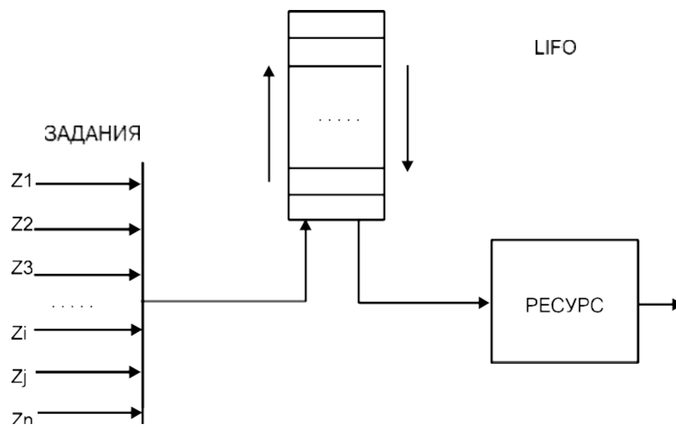


Рис. 4

У цьому режимі важливим є обрання величини кванту часу T. Якщо T має велике значення, тоді алгоритм наближається до алгоритму FIFO. Зі скороченням T зменшується також час обслуговування коротких заявок, але коли величина кванту мала, час перемикання процесора на іншу заявку буде більшим, ніж час обслуговування. Тому необхідно обирати T ні занадто великим, ні занадто малим. Для систем, де передбачається велика кількість користувачів, зменшення кількості заявок може спричинити значну затримку в сатисфакції заявок. Цього можна уникнути за допомогою двох методам.

Перший з них полягає у виборі постійної величини часу циклу, тобто часу очікування для обслуговування заявки користувача з моменту її надходження. Величина T залежить від кількості заявок у черзі; чим більшим буде кількість заявок у черзі, тим меншою буде величина T.

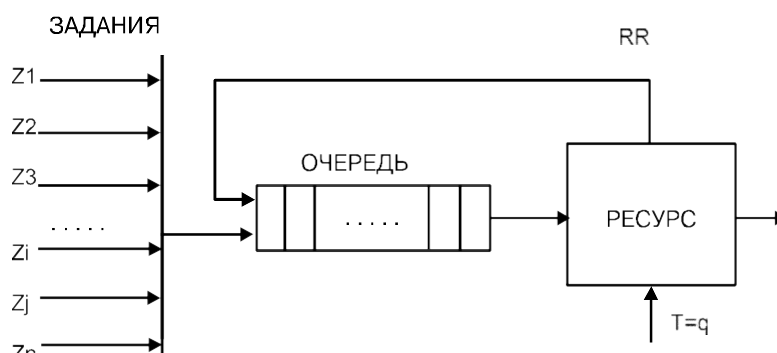


Рис. 5

Другий використовується, коли значне зменшення часу T є небажаним. Для цього визначається мінімальна величина T, і, якщо нові заявки

потребують менший квант, ніж встановлений ліміт, вони не вміщуються у чергу до тих пір, поки будь-яка з заявой у черзі не буде повністю обслугована.

Особливості організації: Заснована на квантуванні. Процес може бути витіснений по закінченню кванта, якщо він до цього часу не закінчився. Він ставиться в кінець черги поряд з надійшли процесами. Черга готових процесів одна - для вже виконувалися і для новоприбулих. З черги завжди вибирається перший процес, ставиться в кінець черги.

Зміна виконуємого процесу може відбутися в наступних випадках:

- процес закінчив своє виконання або відбулася помилка
- процес перейшов у стан очікування.
- закінчився квант часу, що був відведений процесу

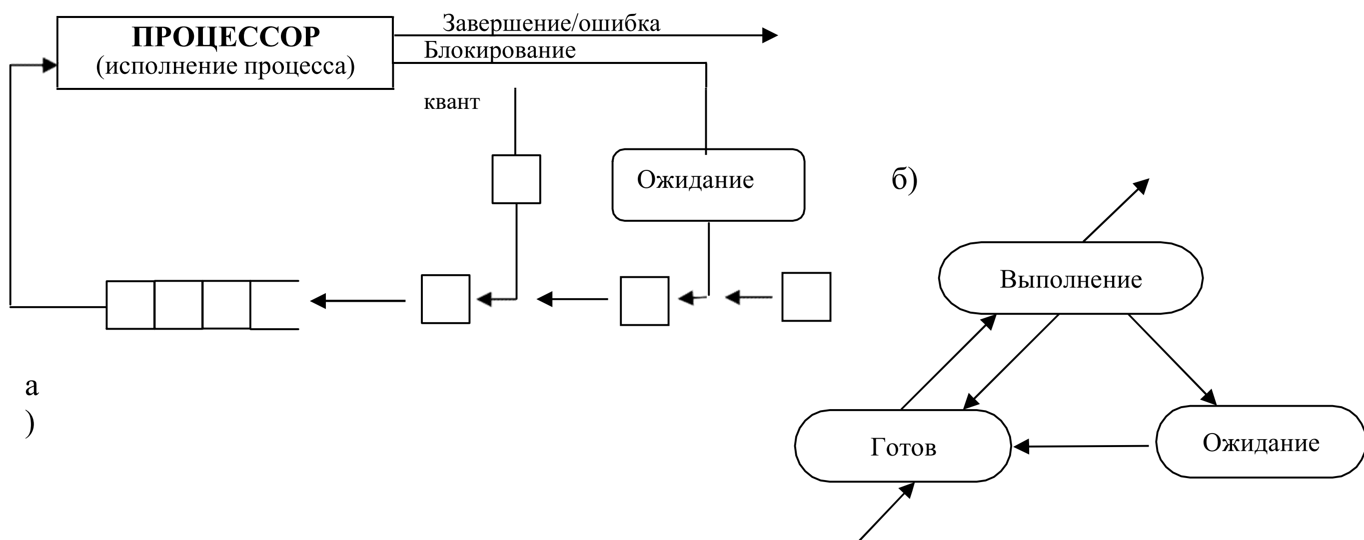


Рис.6. а) Схема циклической дисциплины обслуживания;

б) граф состояний процесса в системе с циклической дисциплиной обслуживания

Багатопріоритетні циклічні алгоритми

Багатопріоритетний циклічний алгоритм, на відміну від попереднього (RR), використовує N черг. Для всіх черг установлюються пріоритети, перша має найбільший. Усередині черги заявки мають рівний пріоритет.

Нова заявка поміщається у чергу з найвищим пріоритетом. Якщо при завершенні кванта обслуговування заявки ресурсом не було завершено, тоді

вона переміщується у кінець черги з меньшим пріоритетом, ніж попередня. Заявки у останній черзі оброблюються без переривання обслуговування.

Цей алгоритм є спрощеним випадком алгоритму Корбато.

Алгоритм Корбато

Вважається, що тривалість виконання програми приблизно пропорційна її довжині. Принаймні, від довжини програми прямо залежить час, що витрачається на передачу програми між ОЗУ і зовнішнім ЗУ при її активізації.

Визначення номера черги, в яку поступає програма при первинному завантаженні, здійснюється по алгоритму планування Корбато: програма відразу поступає в чергу i = [log2 ([lp/ltk] + 1)], де lp - довжина програми в байтах; ltk — кількість байт, які можуть бути передані між ОЗУ і зовнішньою пам'яттю за час tk. [ ] означає цілу частину числа.

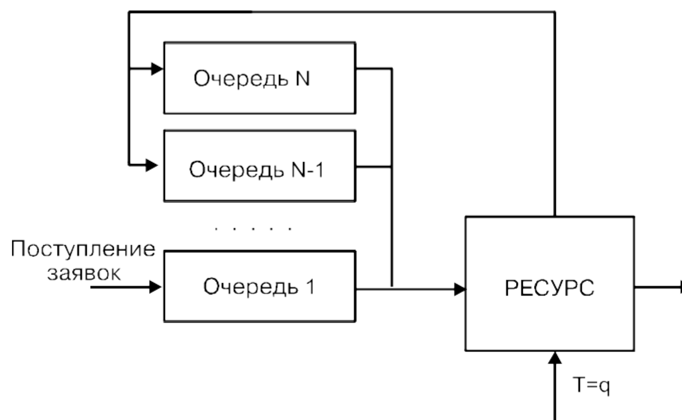


Рис. 7. FBn

Заявці, що надійшла на обслуговування з черги, з номером N призначається квант обслуговування довжиною 2^n *q і, у випадку, коли вона не отримує повне обслуговування, вона надходить до черги n + 1. Якщо протягом часу обслуговування програми у черзі з номером N з'являється програма у черзі N' < N, тоді її обслуговування переривається, вона повертається на початок черги з номером N, починається обслуговування програми у черзі N'.

У циклічних алгоритмах приймається до уваги користь “коротких” програм, що помітно впливає на зменшення величини обміну інформацією між основною і допоміжною пам'яттю.

Ця дисципліна дозволяє скоротити кількість системних перемикань за рахунок того, що програмам , що вимагають більшого часу рішення, надаватимуться чималі кванти часу вже при першому занятті ними ресурсу.

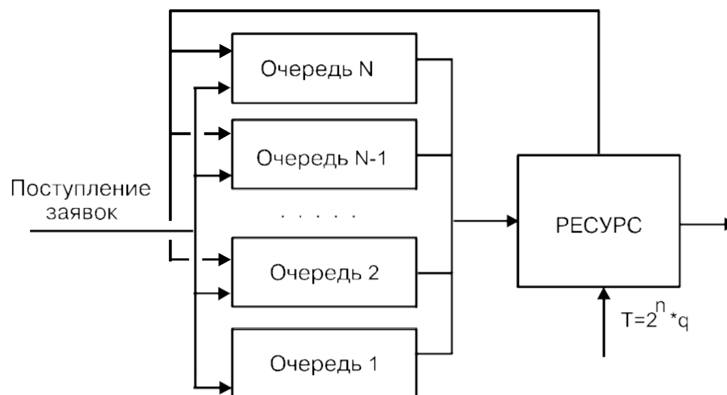


Рис. 8. Алгоритм Корбато

Змішаний алгоритм обслуговування

Змішаний алгоритм обслуговування представляє собою комбінацію простого та багатоприоритетного алгоритмів. Цей алгоритм використовується у системі розподіленого часу ЕОМ AN/FSO-32 фірми IBM та був розроблений як результат експериментальних досліджень.

З програм, які знаходяться в системі, організовані 3 черги. Програми, які введені в систему, розміщуються в першій черзі, де вони мають право на три цикли обслуговування, кожен по 600 мс, відповідно до циклічного алгоритму обслуговування. Програми, які вимагають більше часу обслуговування, переміщуються в другу чергу, де кожна з них отримує обслуговування протягом циклу з шести квантів, кожен з яких 600 мс. Якщо обслуговування ще не закінчене, вони переміщаються в третю чергу.

У цій черзі програма згідно з циклічим алгоритмом отримує обслуговування до кінця і може отримати до 20 квантів по 600 мс кожен раз. Після закінчення кожного кванта обслуговування в чергах 2 і 3 може перериватися, якщо протягом цього часу з'являється програма в черзі 1.

Послідовність в обслуговуванні черг встановлюється відповідно до циклічного багатопріоритетного алгоритму.

3. Дисципліни пріоритетного обслуговування

Реальні властивості функціонування обчислювальних систем в різних завданнях можуть відрізнятися своєю відносною важливістю і тривалістю виконання. Ці обставини змушують розробляти алгоритми різних дисциплін обслуговування, які присвоюють пріоритет завданням більшої важливості до вступу їх в систему. Відповідно до принципів призначення пріоритетів та їх обробки, розрізняють дисципліни з фіксованими і динамічними пріоритетами.

Більш загальні — це дисципліни з відносно фіксованими пріоритетами для різноманітних потоків заявок. Особливістю, яка відрізняє цю дисципліну, є те, що поява заявки з більш високим пріоритетом не викликає переривання обслуговування заявок з меншим пріоритетом.

В алгоритмах з відносними пріоритетами обслуговування кожної нової заявки може початися, як тільки припиниться обслуговування попередньої заявки, не дивлячись на те, що вона має більш низький пріоритет. Як результат зазначеного обмеження, час очікування в черзі для заявок з великим пріоритетом може виявитися дуже великим.

Зменшення часу очікування заявок з великим пріоритетом можна досягти введенням, так званих, абсолютних пріоритетів обслуговування. Обслуговування заявок з низьким пріоритетом переривається кожен раз, коли в системі з'являється заявка з більш високим пріоритетом. Заявки з однаковим пріоритетом отримують обслуговування відповідно до порядку їх появи в системі. Час очікування в черзі заявок з низьким пріоритетом, при наявності переривання, залежить також від методу відновлення перерваного обслуговування.

Дисципліни бувають з відновленням кванта обслуговування і з втратою перерваного кванта обслуговування. У разі використання алгоритмів обслуговування з абсолютними пріоритетами виникає проблема визначення

доцільності переривання заявок з низькими пріоритетами заявками з високими абсолютними пріоритетами. Щоб оцінити ефективність використання абсолютних пріоритетів необхідно визначити загальні втрати, з урахуванням системи штрафів за очікування заявок кожного пріоритету. Система повинна оцінити втрати часу на переривання та відновлення заявки з низьким пріоритетами час, необхідний на її дообслуговування.

Згадані труднощі зумовлюють необхідність шукати деяку проміжну дисципліну серед абсолютних і відносних пріоритетів - адаптивне обслуговування. У разі цієї дисципліни обслуговування визначається доцільність переривання обслуговування заявок з низьким пріоритетом, коли з'являються заявки з високим пріоритетом. Якщо заявка отримала необхідне обслуговування майже повністю, тоді виявляється доцільним не переривати її і завершити обслуговування

Критерій необхідності припинення обслуговування заявок з низьким пріоритетом може бути представлений наступним співвідношенням:

$$\Delta T_{JI} < \frac{\Delta T_I}{a_I} a_J$$

Де:

- **T_{JI} -** час, необхідний для припинення обслуговування заявки j у той момент, коли з'являється заявка з більш високим пріоритетом і;
- a_J, a_I - штраф для кожної одиниці часу очікування заявок і, j;
- T_I - час обслуговування заявки і.

Дисципліна обслуговування з відносними пріоритетами

Особливості організації: Дисципліна обслуговування, заснована на пріоритетах. Процес не може бути витіснений іншими завданнями. Черга готових процесів одна. Процес завжди ставиться в кінець черги. На виконання з черги вибирається процес з найбільшим пріоритетом.

Зміна виконуємого завдання відбувається в наступних випадках:

- процес завершено або сталася помилка;

- процес перейшов у стан очікування.

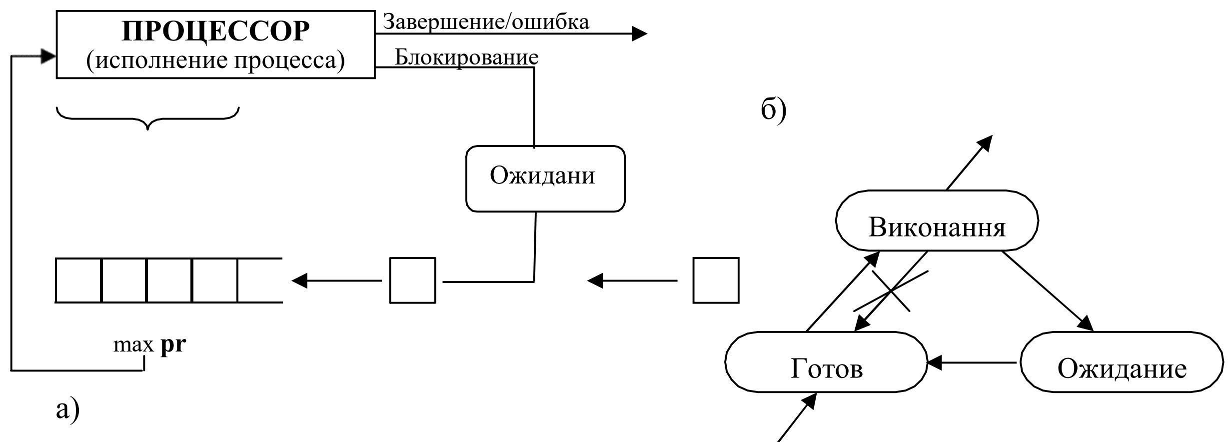


Рис.9. а) Схема дисципліни обслуговування з відносними пріоритетами;

б) Граф станів процесу в системі з дисципліною обслуговування з відносними пріоритетами

Дисципліна обслуговування з абсолютними пріоритетами

Особливості організації: Дисципліна обслуговування, заснована на пріоритетах. Процес може бути витіснений процесом з великим пріоритетом. Процеси ставляться в кінець єдиною черги. З черги вибирається процес з максимальним пріоритетом

Зміна виконуємого завдання відбувається в наступних випадках:

- процес завершений або відбулася помилка;
- процес перейшов у стан очікування;
- у черзі з'явився процес з більшим пріоритетом.

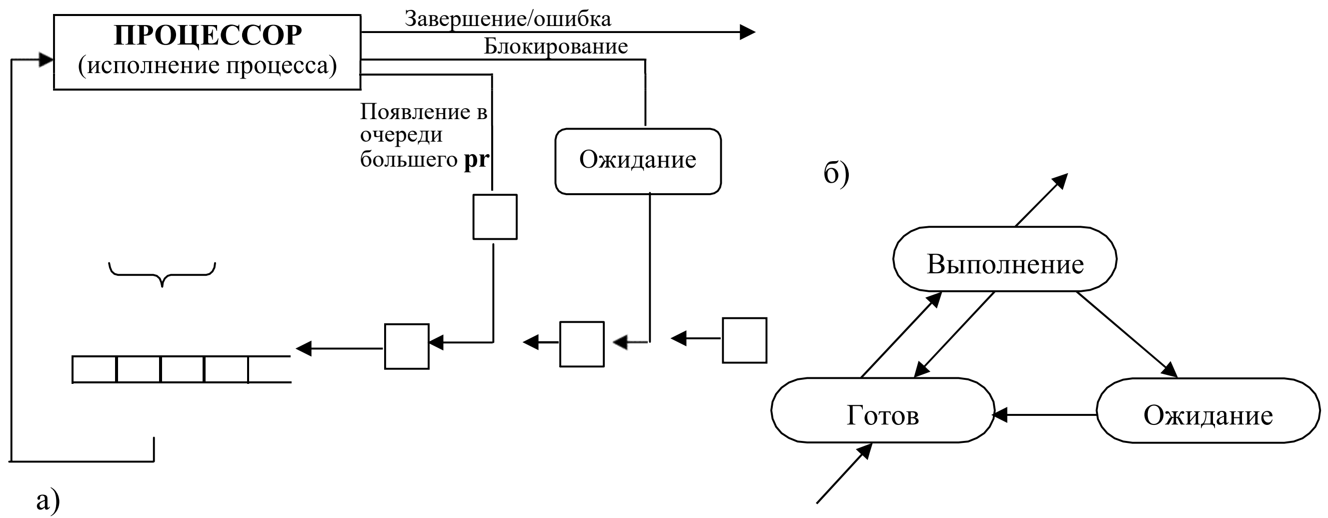


Рис.10. а) Схема дисципліни обслуговування з абсолютними пріоритетами;

б) Граф станів процесу в системі з дисципліною обслуговування з абсолютними пріоритетами

Дисципліна адаптивного обслуговування

<u>Особливості організації:</u> Одночасне використання і пріоритету, і кванта. Алгоритм заснований на квантуванні, але величина кванта залежить від пріоритету: чим вище пріоритет, тим більший проміжок часу процес може займати процесор. Вибір з черги може здійснюватися по першому елементу або відповідно до пріоритетом.

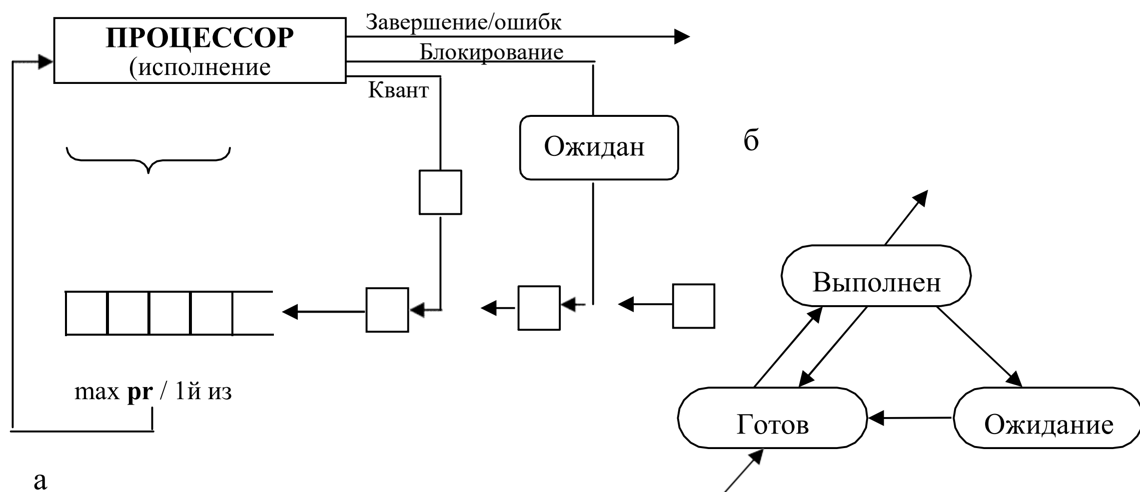


Рис.11.а) Схема дисципліни адаптивного обслуговування;

б) Граф станів процесу в системі з дисципліною адаптивного обслуговування

4. Дисципліни з динамічними пріоритетами

Дисципліни обслуговування з фіксованими пріоритетами передбачають незмінність пріоритетів заявок протягом часу очікування та їх обробки системою.

Так як відносна важливість заявок може змінюватися в залежності від часу очікування обслуговування і зміни часу обслуговування, з'являється необхідність мати такі дисципліни з пріоритетами в обслуговуванні, де пріоритет заявок залежить від реального часу обслуговування або від тривалості очікування заявки в пам'яті машини.

В описі алгоритму обслуговування в ряді випадків доцільно використовувати поєднання декількох дисциплін обслуговування, які призводять до раціонального компромісу серед достоїнств і недоліків для створення кожної з них.

Проаналізуємо зміну часу очікування в черзі в залежності від використання різних дисциплін обслуговування заявок. Зіставляючи час очікування заявок різних пріоритетів, можна показати, що введення відносних пріоритетів має як результат зменшення часу очікування заявок з високим пріоритетом і збільшення часу очікування заявок з низьким пріоритетом в порівнянні з обслуговуванням без пріоритетів (FIFO). Це представлено на рис. 4.10, де крива OP відповідає дисципліні з відносними пріоритетами.

Результат використання пріоритетних дисциплін обслуговування представлений на рис. 4.13, де OP є кривої відносного пріоритету; PA - крива абсолютного пріоритету. Привласнюючи заявці абсолютний пріоритет, ми отримуємо зменшення часу очікування заявок з високими пріоритетами, але одночасно збільшення часу очікування заявок з низькими пріоритетами.

У деяких системах необхідно здійснювати суворі обмеження в часі очікування деяких заявок, фактично необхідно присвоювати зазначеним заявками абсолютні пріоритети. Як результат, часи очікування таких заявок, які мають низькі пріоритети, можуть виявитися високими. Для здійснення

обмежень всіх типів заявок можна разом з абсолютними пріоритетами привласнювати деяким заявками відносні пріоритети, а решта обслуговувати без пріоритетів. Така дисципліна обслуговування називається змішана. Наприклад, в систему надходить М типів заявок. Якщо заявки типу 0-М мають безпріорітетное обслуговування (FIFO), середній час очікування деяких заявок може виявитися неприпустимо велика (рис. 4.12., 4.13.).

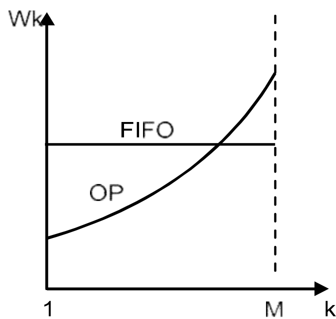


Рис. 12

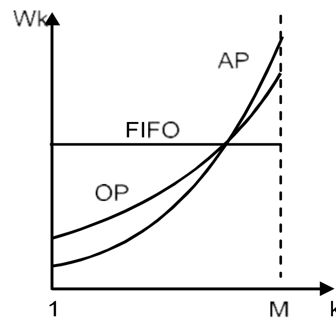


Рис. 13

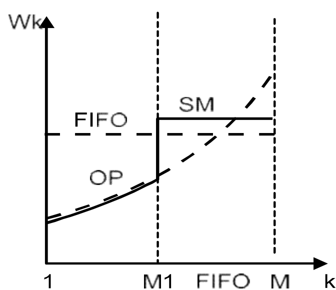


Рис. 14

Якщо заявки отримують обслуговування з відносними пріоритетами (крива OP), тоді для заявок типу 1, .., M1 ситуація поліпшується, але час очікування заявок з низькими пріоритетами перевищить допустимі значення.

Якщо заявки типу 1, .., M1 отримують обслуговування дисципліною з відносним пріоритетом і заявки M1 +1 .. M отримують обслуговування без пріоритету (FIFO), тоді середній час очікування, відповідне кривої SM, може відповідати даним обмеженням (рис. 4.14). Інші випадки використання змішаних дисциплін обслуговування показані на рис. 4.15 і 4.16.

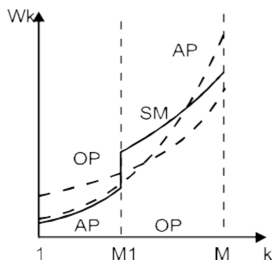


Рис. 15

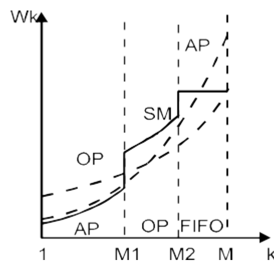


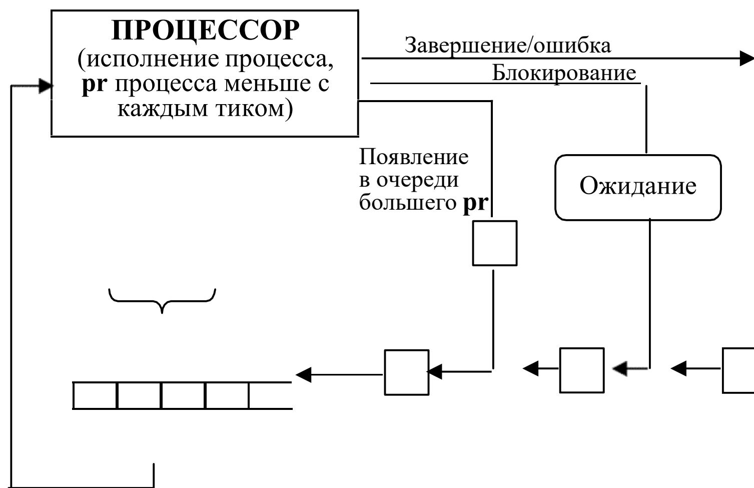
Рис. 16

Дисципліна обслуговування з пріоритетом, що залежить від часу обслуговування

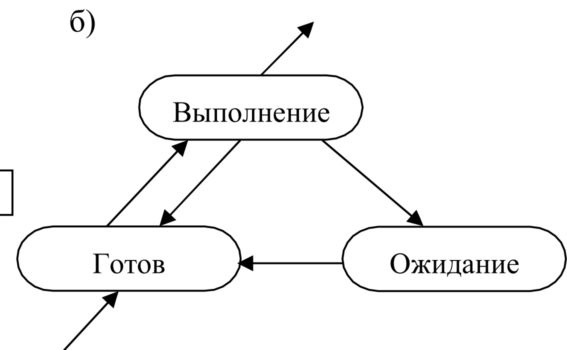
Особливості організації Дисципліна обслуговування, заснована на абсолютних пріоритетах. Під час виконання процесу його пріоритет зменшується з кожним тиком або квантом. Якщо пріоритет процесу стає менше пріоритету процесу стоїть в черзі готових, процес буде витіснений з виконання. Це дозволяє зменшити дискримінацію процесів, що виникає при використанні дисциплін обслуговування з абсолютними пріоритетами.

Зміна виконуємого завдання відбувається в наступних випадках:

- процес завершений або відбулася помилка;
- процес перейшов у стан очікування;
- пріоритет завдання стає меншим, ніж у очікуємого у черзі готових завдання з найбільшим пріоритетом;
- у чергу надійшов процес з більшим пріоритетом.



а)



б)

Рис.8. а) Схема дисципліни обслуговування з пріоритетами, залежними від часу виконання;

б) Граф станів процесу в системі з дисципліною обслуговування з пріоритетами, залежними від часу виконання

Дисципліна обслуговування з пріоритетом, що залежить від часу очікування у черзі готових процесів

Особливості організації: Дисципліна обслуговування, заснована на абсолютних пріоритетах. У міру очікування в черзі готових, пріоритет процесу збільшується з кожним тиком або квантом. Якщо пріоритет виконується процесу стає менше пріоритету процесу стоїть в черзі готових (незалежно від того, новий це процес або давно стоїть), процес буде витіснений з виконання. Це дозволяє виключити дискримінацію процесів, що виникає при використанні дисциплін обслуговування з абсолютними пріоритетами і дисциплін обслуговування з пріоритетами, залежними від часу виконання.
Зміна виконуємого завдання відбувається в наступних випадках:

- процес завершений або відбулася помилка;
- процес перейшов у стан очікування;
- пріоритет виконуємого завдання стає меншим, ніж у тієї заявки, що очікує у черзі готових з найбільшим пріоритетом;
- у чергу надійшов процес з більшим пріоритетом.

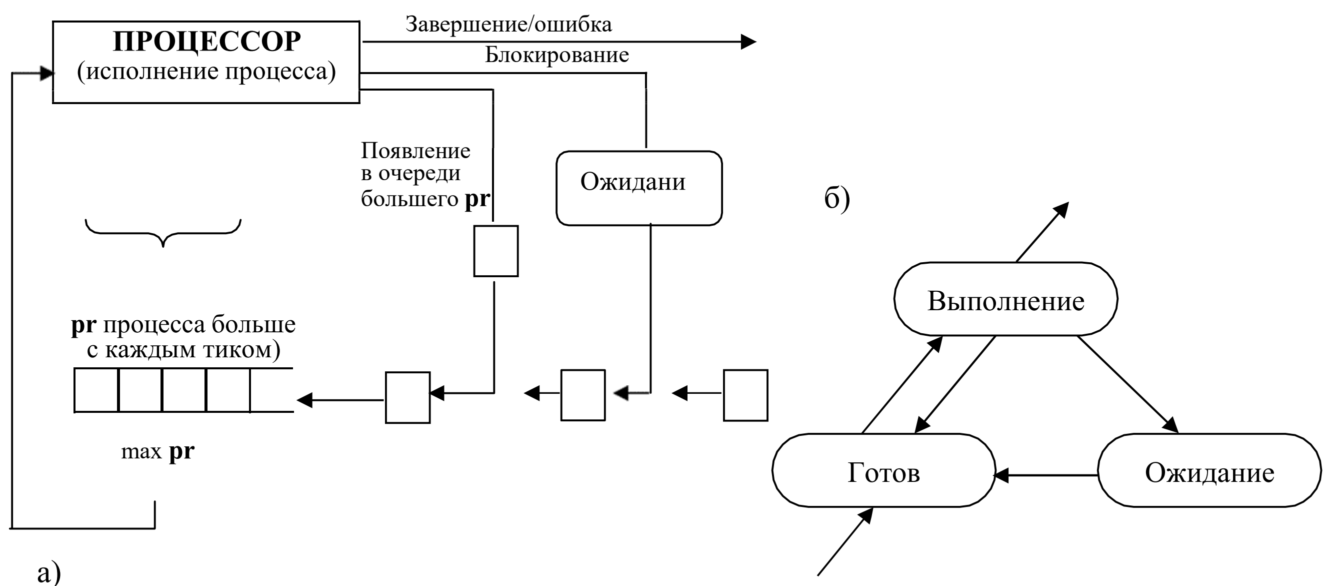


Рис.9. а) Схема дисципліни обслуговування з пріоритетами, залежними від часу очікування;

б) Граф станів процесу в системі з дисципліною обслуговування з пріоритетами, залежними від часу очікування

1. Варіанти

Варіант вибирається за двома останніми номерами залікової книжки по mod 11 + 1

1. Алгоритм FIFO (*First Input- First Output*)
2. Алгоритм LIFO
3. Алгоритм випадкового вибору заявок усередині черги
4. Алгоритм RR (Round Robin)
5. Алгоритм Fbn (Foregraund -Backgraund)
6. Змішаний алгоритм
7. Алгоритм Корбато
8. Алгоритм обслуговування з відносним пріоритетом
9. Алгоритм обслуговування з абсолютним пріоритетом
10. Алгоритм адаптивного обслуговування
11. Алгоритм з залежністю приоритету від часу очікування
12. Алгоритм з залежністю приоритету від часу обслуговування

* У всіх варіантах кількість черг та кількість пріоритетів задаються параметрами.

1 Звіт повинен мати:

1 Опис роботи дисципліни обслуговування.

2 Лістинг основної частини програми.

3 Пояснення форм графіків.

4 Переваги та недоліки досліджуваної дисципліни обслуговування.

Електронна версія (мережа КПІ)

ftp://vt513.comsys.ntu-kpi.kiev.ua/pub/edu/spo1/spo1-3.doc

ОС. Лабораторна робота № 4

Найпростіший синтаксичний аналізатор (парсер)

Коротка теорія

Задача синтаксичного аналізатора (парсера) полягає у перевірці відповідності вхідного файлу допустимому формату і у виявленні з символів вхідного файлу синтаксичних конструкцій. Входом для парсера є потік символів і визначення формату синтаксису. Виходом для парсера є розпізнані синтаксичні конструкції.

Парсер можна побудувати декількома способами. Один із способів - це розбір вхідного файлу за допомогою автомата. Символ із вхідного файлу і поточний стан автомата визначають дію і наступний стан автомата. Цей процес можна відобразити наступним чином:

Автомат (символ, стан) → (дія, стан)

Перевага такого методу побудови парсера в тому, що швидкість розбору вхідного файлу не залежить від складності перевіряємого формату. Логіка

роботи автомата однакова як для простих, так і для складних форматів. Застосування автоматів дозволяє створювати уніфікований інтерфейс для розбору файлів.

Автомат можна представити таблицею. Координата стовпця таблиці відповідає номером стану автомата, а координата рядка таблиці відповідає коду символу. Немає необхідності створювати рядки для кожного можливого символу, так як деякі символи вимагають однакової обробки. Для цього кожному символу призначається клас символу (деяке число). Метод перекодування символів у класи символів залежить від кількості класів.

Якщо синтаксис вхідного файлу призводить до створення великого автомата, то замість одного автомата можна побудувати дерево автоматів. Спочатку створюються і відлагоджуються невеликі автомати для розбору деяких частин синтаксису. Потім створюється головний автомат, який зв'язує в дерево всі автомати. Використання дерева автоматів дозволяє використовувати вже готові автомати при побудові нових.

Завдання на роботу

Розробити інфраструктуру (структури даних, константи, набір функцій, методів, класів, порядок роботи і т. п.) для побудови парсера на основі роботи автоматів.

Необхідні умови:

a. Символи вхідного файлу можуть бути в будь-якому кодуванні (один символ займає не обов'язково 1 байт), вхідний файл складається з будь-якої кількості символів;

b. Один парсер може складатися з декількох автоматів, завжди є один головний автомат, з головного автомата можна переходити в інші автомати парсера

і т. д. вниз по дереву;

c. Кожен автомат управляється одним вхідним символом, поточним станом автомата і можливо поточним контекстом розбору;

d. Повинна бути функція / метод отримання чергової синтаксичної кон-струкції з вхідного файлу або отримання інформації про помилку розбору.

Використовуючи розроблену інфраструктуру створити парсери для наступних вхід-них файлів:

1. Вхідний файл складається з email адрес розділених пробілами або символами перекладу рядка, синтаксична конструкція - це ім'я і домен. Email адреса складається з імені, символу '@' і доменного імені. Ім'я складається з букв англійського алфавіту, цифр та символу підкреслення. Доменне ім'я складається з одного або декількох імен піддоменів, розділених символом '.' (Крапка). Ім'я піддомену складається з однієї або декількох букв англійського алфавіту, цифр та символу "-" (мінус), повинно починатися і закінчуватися буквою або цифрою.

2. Вхідний файл має наступний синтаксис:

a. Вхідний файл складається з коментарів, секцій та параметрів, використовуються ASCII символи.

b. Коментар може починатися з символу '#' і закінчуватися символом нового рядка або кінцем файлу.

c. Коментар може починатися з послідовності символів / * та за-закінчуються послідовністю символів * /.

d. Коментарі можуть бути розташовані в будь-якому місці вхідного файлу.

e. Параметр починається з імені параметра, далі йде символ '=', потім значення параметра, значення параметра завершується символом ';'.

f. Секція починається з імені секції, далі йде опціональне значення, яке завершується символом "{', секція завершується символом'} '. Всередині секції можуть бути коментарі, параметри і секції.

g. Значення можуть складатися з будь-яких символів, розділених пробілами і символами нового рядка.

h. Значення можуть містити рядки, укладені в подвійні лапки. Подвійні лапки в рядку кодуються двома символами \ ", символ '\' кодується - \ \, символ нового рядка кодується - \ n і символ табуляції кодується - \ t. Будь-який інший символ після '\' вважається помилкою.

i. Праворуч і ліворуч від '=' після імені параметра, праворуч і ліворуч від ';' в кінці значення параметра, праворуч і ліворуч від "{'i'} ' в секції і між ім'ям секції і опціональним значенням допускається будь-яка кількість пробілів символів нового рядка.

j. Формат значень не обмовляється. Ім'я параметра або секції повинно починатися з літери, далі можуть йти букви, цифри і символ підкреслення.

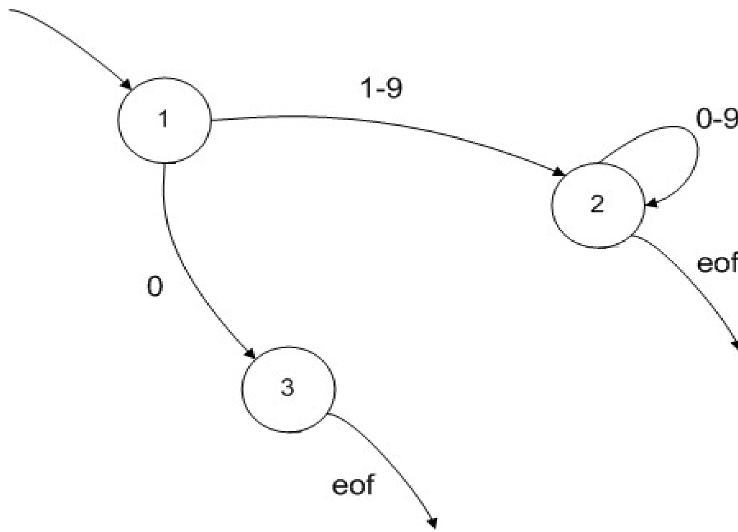
Розбір синтаксису в пунктах b, c, h необхідно реалізувати в окремих автоматах.

Синтаксична конструкція - це тип конструкції (початок секції, кінець секції, параметр) і, залежно від типу конструкції, ім'я секції плюс опціональне значення, нічого, ім'я параметра плюс значення.

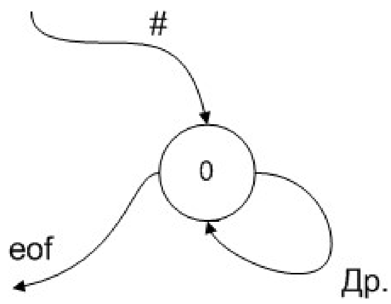
Якщо в значенні присутні кілька пробілів, символів табуляції або символів перекладу рядка, тоді їх необхідно замінити одиночним пропуском.

Приклади:

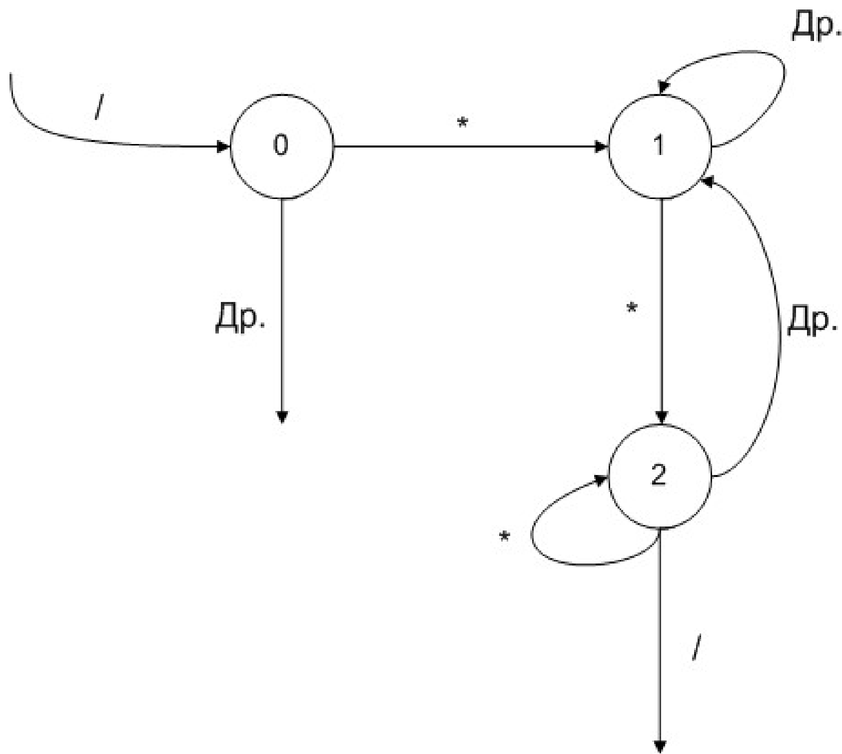
1) Граф автомата для одного цілого десяткового числа:



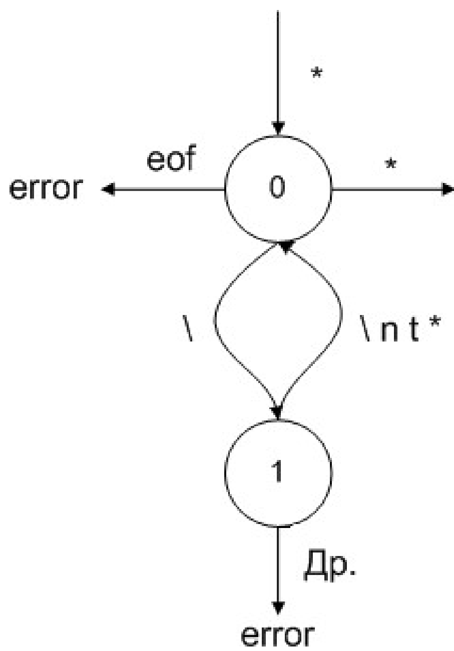
2) Граф автомата для однорядкового коментаря:



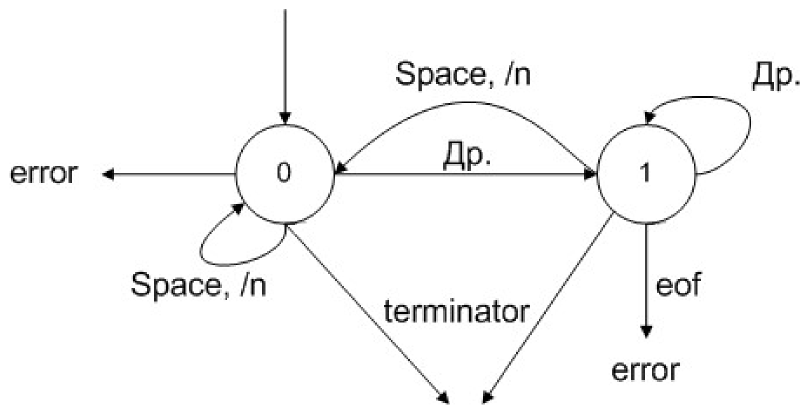
3) Граф автомата для багаторядкового коментаря:



4) Граф автомата для рядка:



5) Граф автомата для значення (як узагальнення замість “;” чи “{“ в якості завершувача значення використовується “terminator”):



Обмеження при реалізації

При виконанні цієї роботи не можна використовувати генератори лексичних і синтаксичних аналізаторів, регулярні вирази і т. п.

Звіт

Звіт повинен містити:

1. Опис ідеї розробленої інфраструктури.
2. Графічне представлення розробленої інфраструктури.
3. Графічне подання автоматів для кожного синтаксису.
4. Лістинг розробленої інфраструктури.
5. Лістинг двох парсерів.
6. Приклад роботи парсера для кожного синтаксису.

Електронна версія(мережа КПІ)

ftp://vt513.comsys.ntu-kpi.kiev.ua/pub/edu/spo2/lab2-1.doc

Література

1. Ахо, Сети, Ульман. *Компиляторы: принципы, технологии и инструменты*.

Лабораторна робота №5

АЛГОРИТМИ ЗАМІЩЕННЯ СТОРІНОК ВІРТУАЛЬНОЇ ПАМ'ЯТІ

Короткі теоретичні відомості

Однією із задач віртуальної пам'яті є надання програмам адресних просторів пам'яті, що перевищують обсяг фізичної пам'яті. Така організація роботи з пам’яттю досягається шляхом створення карт відображення віртуальних сторінок у фізичні сторінки для кожного процесу. Одна сторінка віртуальної відображається в одну або декілька фізичних сторінок. Карта відображення віртуальних сторінок у фізичні сторінки складається із записів. Кожен запис має містити, як мінімум, адресу фізичної сторінки, права доступу та біт присутності. Зазвичай кожен запис також містить біт модифікації і біт звернення.

Так як кількість сторінок фізичної пам'яті менше ніж кількість сторінок віртуальної пам'яті, виникає необхідність заміщення відображених сторінок сторінками необхідними для продовження роботи процесів. Складність прийняття рішення, яку сторінку можна перемістити в зовнішній файл (своп), викликана тим, що інформація про подальші зверненнях до пам'яті процесами відсутня. Однак можна побудувати ефективні алгоритми заміщення сторінок, оскільки робота процесів характеризується принципом локальності: в деякий проміжок часу будь-який процес звертається до обмеженого набору сторінок, які складають робочий набір у цей проміжок часу.

Алгоритм заміщення сторінок накопичує статистику звернення до пам'яті (досліджуються біт обігу та/або біт модифікації кожної сторінки) і при необхідності вибирає "саму непотрібну сторінку", переміщує її вміст у зовнішній файл (своп) і віддає звільнену фізичну сторінку для нового відображення.

Щоб дати можливість операційній системі збирати корисні статистичні дані про те, які сторінки використовуються, а які - ні, більшість комп'ютерів з

віртуальною пам'яттю підтримують два статусних бита, пов'язаних з кожною сторінкою. Біт R (від Referenced - звернення) встановлюється кожен раз, коли відбувається звернення до сторінки (читання або запис). Біт M (від Modified - зміна) встановлюється, коли сторінка записується (тобто змінюється).

Алгоритм NRU

Алгоритм NRU (Not Recently Used) видаляє сторінку за допомогою випадкового пошуку в не порожньому класі з найменшим номером. Мається на увазі, що краще вивантажити змінену сторінку, до якої не було звернень, хоча б протягом одного такту системних годин (зазвичай 20 мс), ніж стерти часто використовувану.

Коли процес запускається, обидва сторінкових біта (R і M) для всіх його сторінок операційною системою скинуті в 0. Періодично (наприклад, при кожному перериванні по таймеру) біт R очищається з метою відрізнити сторінки, до яких давно не було звернення, від тих, на які посилання були.

При порушенні сторінкового переривання операційна система перевіряє всі сторінки і ділить їх на чотири категорії на підставі поточних значень бітів R і M:

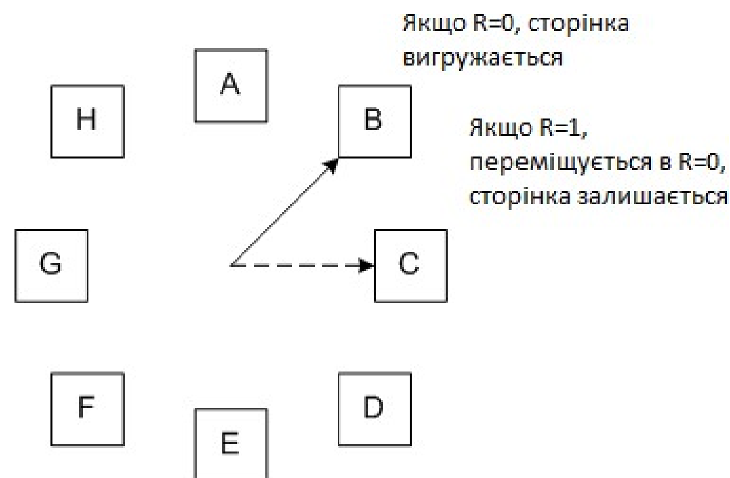
- клас 0: не було звернень і змін (R = 0, M = 0);
- клас 1: не було звернень, сторінка змінена (R = 0, M = 1);
- клас 2: було звернення, сторінка не змінена (R = 1, M = 0);
- клас 3: сталося і звернення, і зміна (R = 1, M = 1).

Хоча клас 1 на перший погляд здається неможливим, таке трапляється, коли у сторінки з класу 3 біт R скидається під час переривання по таймеру. Переривання по таймер не затирають біт M, оскільки ця інформація необхідна для того, щоб знати, чи потрібно переписувати сторінку на диску чи ні. Тому якщо біт R встановлюється на нуль, а M залишається недоторканим, сторінка потрапляє в клас 1.

Алгоритм «годинник»

Щоб уникнути переміщення сторінок по списку, можна використовувати покажчик, який переміщається по списку.

Коли відбувається сторінкове переривання, перевіряється та сторінка, на яку спрямована стрілка. Якщо її біт R дорівнює 0, сторінка вивантажується, на її місце в часовому колі стає нова сторінка, а стрілка зсувається вперед на одну позицію. Якщо біт R дорівнює 1, він скидається, стрілка переміщається до наступної сторінки. Цей процес повторюється до тих пір, поки не знаходиться та сторінка, у якій біт R = 0.



Алгоритм старіння "aging", модифікований NFU.

Алгоритм старіння є модифікацією алгоритму NFU, який, у свою чергу, є різновидом алгоритму LRU. Змінення зводяться до двох модифікацій. По-перше, кожен лічильник зсувається вправо на один розряд перед додаванням біта R. По-друге, біт R вдвигается в крайній зліва, а не в крайній праворуч розряд лічильника.

	Біти R для сторінок 0-5, такт 0	Біти R для стрінок 0-5, такт 1	Біти R для стрінок 0-5, такт 2	Біти R для стрінок 0-5, такт 3	Біти R для стрінок 0-5, такт 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
Сторінка					
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	а	*б*	*в*	*г*	*д*

На рисунку продемонстровано, як працює алгоритм «старіння». Припустимо, що після першого тіку годин біти R для сторінок від 0 до 5 мають значення 1, 0, 1,0, 1, 1 відповідно (у сторінки 0 біт R дорівнює 1, у сторінки 1 - R = 0, у сторінки 2 - R = 1 і т. д.). Іншими словами, між тіком 0 і тіком 1 відбулося звернення до сторінок 0, 2, 4 і 5, їх біти R взяли значення 1, решта зберегли значення 0. Після того як шість відповідних лічильників зрушилися на розряд і біт R зайняв крайню зліва позицію, лічильники отримали значення, показані на рис. а. Інші чотири колонки рисунка зображують шість лічильників після наступних чотирьох тіків годин.

Коли відбувається сторінкове переривання, віддаляється та сторінка, лічильник якої має найменшу величину. Ясно, що лічильник сторінки, до якої не було звернень, скажімо, за чотири тіка, буде починатися з чотирьох нулів і, таким чином, мати більш низьке значення, ніж лічильник сторінки, на яку не посилалися протягом лише трьох тіків годин.

У цьому алгоритмі лічильник має кінцеву кількість розрядів.

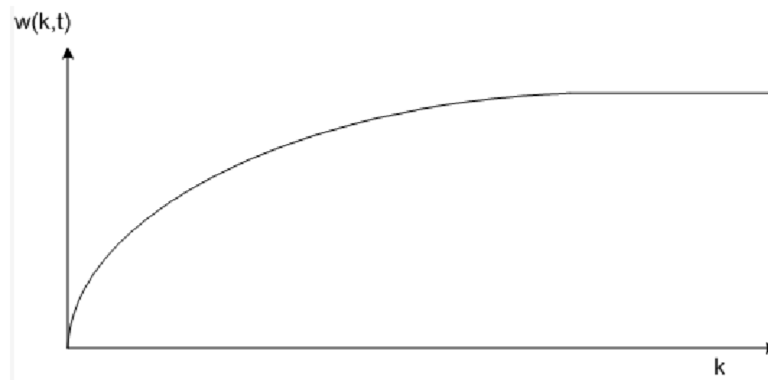
Алгоритм "робочий набір"

Заміщення сторінок за запитом - коли сторінки завантажуються на вимогу, а не заздалегідь, тобто процес переривається і чекає завантаження сторінки.

Буксування - коли кожну наступну сторінку доводиться процесу завантажувати в пам'ять.

Щоб не відбувалося частих переривань, бажано щоб часто запитувані сторінки завантажувалися заздалегідь, а решта довантажувати за потребою.

Множина сторінок (k), яку процес використовує в даний момент (t), називається *робочим набором*. Тобто можна записати функцію w (k, t).



Тобто робочий набір виходить в насичення, значення w (k, t) в режимі насичення може служити для робочого набору, який необхідно завантажувати до запуску процесу.

Алгоритм полягає в тому, щоб визначити робочий набір, знайти і вивантажити сторінку, яка не входить в робочий набір.

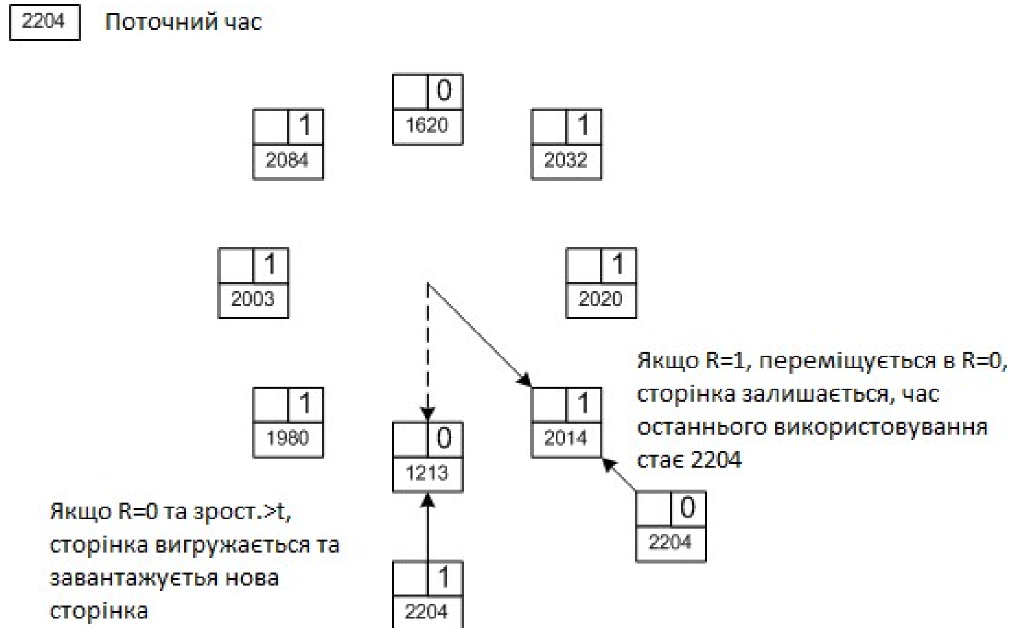
Для реалізації моделі робочого набору необхідно, щоб операційна система відстежувала, які сторінки в ньому знаходяться. Один зі способів отримати таку інформацію - використовувати описаний вище алгоритм старіння. Нехай встановлений старший біт лічильника сторінки означає, що вона входить в робочий набір. Якщо протягом n послідовних тактів годин до такої сторінці не було зроблено звернень, вона викидається з робочого набору. Параметр n доведеться визначити експериментальним шляхом, але продуктивність системи зазвичай не надто чутлива до його точного значення.

Алгоритм WSClock

Алгоритм заснований на алгоритмі "годинник", але використовує робочий набір. Зазвичай, коли «стрілка годинника» показує на сторінку, у якій біт R дорівнює нулю, ця сторінка видаляється. Щоб поліпшити

алгоритм, можна додатково перевіряти, чи входить сторінка в робочий набір поточного процесу, і якщо так, залишати її.

В алгоритмі використовуються біти R та M, а також час останнього використання.



Вихідні дані

В системі є N сторінок фізичної пам'яті. Кожній віртуальній сторінці відповідає рівно одна фізична сторінка.

В системі є декілька процесів. Кожен процес має власний віртуальний адресний простір, число сторінок в якому може бути як менше, так і більше ніж N. Будь-яка фізична сторінка може бути відображена у віртуальному адресному просторі лише одного процесу.

Кожен процес періодично звертається до якоїсь своєї сторінки. Це може бути як нова сторінка, так і та до якої було звернення в минулий раз, але всі процеси характеризуються локальністю звернень (90/10). Звернення може бути читанням або записом (50/50).

В системі використовується один з алгоритмів заміщення сторінок. Сторінки заміщуються за вимогою (demand paging). У системі є таймер і за необхідності можуть бути запущені додаткові системні процеси, наприклад, фоновий процес для різних перевірок стану пам'яті.

Для кожної віртуальної сторінки в карті відображення є адреса відповідної фізичної сторінки, біт присутності, біт модифікації та біт звернення. Біти прав доступу не використовуються.

У системі присутній своп, який реалізовувати не потрібно, але необхідно розрізняти сторінки, переміщені в своп від сторінок, до яких ще не було звернень.

Завдання

Розробити модель системи із сторінковою організацією пам'яті, що задовольняє наведені вище вихідні дані, з реалізацією одного з алгоритмів заміщення сторінок.

Моделююча програма повинна в стандартний потік виводу і в лог-файл виводити звіт про характеристики віртуальної пам'яті процесів, сторінкових перериваннях, звіт про рішення алгоритму заміщення сторінок і т. п.

Алгоритм заміщення сторінок вибирається за варіантом (визначається за номером залікової книжки, останні_дві_ціфри mod 5 + 1):

1. NRU, сторінка, що останнім часом не використовувалась.
2. Алгоритм "годинник", модифікований алгоритм "друга спроба".
3. Алгоритм старіння "aging", модифікований NFU.
4. Алгоритм "робочий набір".
5. Алгоритм WSClock.

Політику розподілу пам'яті (локальна або глобальна) вибрати самостійно.

Звіт

1. Опис алгоритму заміщення сторінок.
2. Опис структур даних, що представляють віртуальну пам'ять, із зазначенням залежності по даним.
3. Лістинг основної частини програми.
4. Висновки: опис переваг і недоліків реалізованого алгоритму заміщення сторінок.

Електронна версія (мережа КПІ)

ftp://vt513.comsys.ntu-kpi.kiev.ua/pub/edu/spo2/lab2-2.doc

ОС. Лабораторна работа №6

Файловая система (часть 1)

Теория

В данной лабораторной работе необходимо разработать файловую систему для устройств хранения информации блочного типа, например магнитный жёсткий диск. В таких устройствах ввод/вывод происходит поблочно (один или несколько блоков за одно обращение к устройству), размер блока определяется аппаратурой. Обычно в таких устройствах не учитываются ограничения на количество изменений в одном блоке.

В файловой системе есть два типа файлов: обычные и директории. В этой лабораторной работе необходимо реализовать одноуровневую файловую систему с одной директорией, содержащей обычные файлы.

Содержимое каждого файла хранится в блоках, один блок содержит информацию только одного файла. Для учёта занятости блоков необходимо применить битовую карту, один бит на один блок. Так как количество блоков в устройстве постоянно, то битовая карта будет содержать постоянное количество элементов. У некоторых файлов последний блок будет использоваться не полностью под данные файла. Блоки в структурах файловой системы адресуются по их порядковым номерам.

Каждый файл, как объект файловой системы, представлен дескриптором. Количество дескрипторов заранее задаётся, поэтому в файловой системе не может быть создано больше определённого количество файлов, даже если в устройстве есть свободное место. Дескриптор файла должен содержать, по крайней мере, следующую информацию: тип файла (обычный файл или директория), количество ссылок на файл, размер файла, карта расположения блоков файла. Карта расположения блоков имеет следующую структуру: есть несколько прямых ссылок на блоки, есть одна ссылка на один блок, содержащий прямые ссылки на блоки. Номер (позиция) ссылки определяет смещение данных в файле, ссылки нумеруются подряд. Ссылка не указывает на блок в двух случаях: соответствующее смещение превышает размер файла или все байты файла соответствующего блока файла равны нулю. Обычные файлы используются пользователем для хранения информации и для файловой системы не имеют форматов.

Директория это файл, данные которого это массив ссылок на файлы. Формат данных директории задаётся файловой системой и не интерпретируется пользователем. Ссылка на файл это имя файла и соответствующий этому имени номер дескриптора файла. Так как имя файла не является частью дескриптора файла, то на один файл может быть несколько ссылок с разными путевыми именами. Так как ссылки на файлы могут быть уничтожены, то необходимо предусмотреть возможность отметить недействительные ссылки на файлы. Имя файла это набор символов, длина имени файла ограничена форматом.

В этой лабораторной работе не требуется организовывать буферный кеш и кеш имён файлов. Драйвер файловой системы при любом запросе должен работать непосредственно с данными на устройстве хранения информации.

Задание

Разработать драйвер файловой системы для устройства хранения информации блочного типа, используя идею структур данных, описанную выше. В качестве файловой системы взять обычный файл, размер файла определяет объём устройства хранения информации. Самостоятельно выбрать размер блока, количество ссылок на блоки в дескриторе файла, максимальную длину имени файла.

Разработать консольную программу, поддерживающую следующие команды:

- *mount* - подключить файловую систему, сохранённую в файле;
- *umount* - отключить файловую систему, в драйвере не должно остаться каких-либо данных о файловой системе;
- *filestat id* – вывести информацию о дескрипторе файла с указанным номером;
- *ls* - вывести список файлов, с указанием номеров дескрипторов файлов;
- *create имя* - создать файл с заданным именем;
- *open имя* – открыть файл с указанным именем, команда должна назначить уникальный номер *fd*, числовой дескриптор для работы с открытым файлом;
- *close fd* - закрыть ранее открытый файл, уникальный номер *fd* больше не должен быть связан с файлом;
- *read fd смещение размер* – прочитать данные из файла по заданному смещению указанного размера;
- *write fd смещение размер* – записать данные в файл по заданному смещению указанного размера;
- *link имя1 имя2* - создать ссылку с именем *имя2* на существующий файл с именем *имя1*;
- *unlink имя* - уничтожить ссылку с заданным именем;
- *truncate имя размер* – изменить размер файла, если размер файла увеличивается, то неинициализированные данные равны нулю.

Отчёт

Отчёт должен содержать:

1. Описание идеи реализации файловой системы.

2. Описание форматов структур файловой системы.

3. Листинг основной части разработанной программы.

Электронная версия (сеть КПИ)
ftp://vt513.comsys.ntu-kpi.kiev.ua/pub/edu/spo2/spo2-3.doc

Литература

1. А. Таненбаум *Современные операционные системы.*

2. Uresh Vahalia *Unix Internals: The New Frontiers.*

ОС. Лабораторна работа №7

Файловая система (часть 2)

Теория

В данной лабораторной работе необходимо продолжить работу над файловой системой из лабораторной работы №3 и реализовать дерево директорий и поддержку символических ссылок.

Дерево директорий позволяет организовывать ссылки на файлы в иерархическую структуру и строится при помощи создания ссылок на директории. Корневая директория или начальная директория дерева имеет фиксированный номер дескриптора, который определяется форматом файловой системы. Остальные директории имеют произвольные номера дескрипторов. Ссылки на обычные файлы и ссылки на директории имеют один и тот же формат в массиве ссылок на файлы, отличие заключается в типе файла на который указывает ссылка. Любая директория всегда имеет две предопределённые ссылки "." и "..", которые указывают на текущую и родительскую директорию. Ссылка ".." в корневой директории указывает на саму директорию.

Имя файла может быть абсолютным или относительным. Абсолютное имя файла это путевое имя файла с указанием пути к файлу начиная с корневой директории. Относительное имя файла это имя файла с указанием пути к файлу начиная с текущей рабочей директории. Каждый процесс (в том числе и командная оболочка) имеет текущую рабочую директорию, которую можно менять. Каждая компонента путевого имени файла, за исключением последней, должна быть ссылкой на директорию. В процессе поиска файла по путевому имени (name lookup) необходимо учитывать такие компоненты имени как ".", ".." и символические ссылки.

Символическая ссылка это тип файла содержимым которого является путевое имя. Если некоторое путевое имя файла содержит компоненту представляющую символическую ссылку, то содержимое символической ссылки необходимо конкатенировать с оставшимися компонентами путевого имени файла. Символическая ссылка может указывать на абсолютное или на относительное путевое имя. В случае абсолютного путевого имени поиск начинается с корневой директории. В случае относительного путевого имени поиск начинается с директории в которой находится символическая ссылка.

Символическая ссылка указывает на файл, но не является ещё одним именем файла. Поэтому создание и уничтожение символической ссылки не приводит изменению счётчика ссылок файла на который указывает символическая ссылка. Создание и уничтожения ссылок (жёстких ссылок) на директории и обычные файлы никаким образом не влияют на символические ссылки, поэтому можно создавать символическую ссылку на произвольное, в том числе и не существующее, путевое имя файла. Тип файла на который указывает символическая ссылка определяется самим файлом.

Символические ссылки могут указывать на другие символические ссылки, поэтому возможно зацикливание при поиске файла по имени. Для предотвращения возможного зацикливания при поиске файла по имени необходимо задать максимально возможное количество переходов по символическим ссылкам для одного путевого имени.

Задание

Добавить в драйвер файловой системы, разработанный в лабораторной работе №3, поддержку дерева директорий и символических ссылок. Добавить в консольную программу команды для работы с директориями и символическими ссылками.

Консольная программа должна поддерживать путевые имена файлов в командах, файл ищется по путевому имени начиная с текущей рабочей директории. Команды *create*, *open*, *truncate* и *cd* должны работать с файлами на которое указывает имя (если последняя компонента имени файла это символическая ссылка, то необходимо перейти по этой ссылке). Все остальные команды, которые работают с именами файлов, работают с заданными файлами по имени (если последняя компонента имени файла это символическая ссылка, то команда применяется к самой символической ссылки, а не к файлу на который указывает символическая ссылка). Команда *link* должна создавать жёсткие ссылки только для обычных файлов.

Добавить в консольную программу поддержку следующих команд:

- *mkdir имя* – создать директорию с указанным именем;
- *rmdir имя* – уничтожить пустую директорию с указанным именем (директория не должно содержать каких либо ссылок, за исключением предопределённых "." и "..");
- *cd имя* – сменить текущую рабочую директорию;
- *pwd* – вывести текущую рабочую директорию;
- *symlink имя1 имя2* – создать символическую ссылку с именем *имя2* на путевое имя *имя1*.

Отчёт

Отчёт должен содержать:

4. Описание идеи реализации файловой системы.

5. Описание форматов структур файловой системы.

6. Листинг основных изменений в разработанной программе.

Электронная версия (сеть КПИ)
ftp://vt513.comsys.ntu-kpi.kiev.ua/pub/edu/spo2/spo2-4.doc
Литература

3. А. Таненбаум *Современные операционные системы.*

4. Uresh Vahalia *Unix Internals: The New Frontiers.*

ОС. Лабораторні роботи №8

Файлова система для флеш-пам’яті типу NAND

Теорія

Флеш-пам'ять типу NAND (далі «флеш-пам'ять») має властивості, які відрізняють цей тип пам'яті від оперативної пам'яті, магнітних дисків і CD-ROM.

Флеш-пам’ять складається з блоків, кожен блок складається із сторінок (зазвичай розміром 512 байт), кожна сторінка може мати до 16 байт додаткової пам’яті, яку можна використовувати для зберігання контрольної суми.

Читання даних з флеш-памяті відбувається посторінково, обмежень на кількість зчитувань даних з однієї сторінки немає. Зміна одного біта зі значення 1 в значення 0 називається програмуванням (programming). Програмування флеш-пам’яті відбувається посторінково. Деякі моделі флеш-пам’яті дозволяють програмувати дані в одній сторінці до 10 разів, після чого вміст сторінки стане невизначеним. Змінити один біт із значення 0 в значення 1 неможливо, але можна очистити (erase) увесь блок, тим самим встановивши значення усіх бітів блоку в 1. Кількість циклів запису-очищення обмежена і дорівнює 100.000 для більшості моделей флеш-пам’яті.

Файлові системи, розроблені для магнітних дисків, не можна застосовувати для флеш-пам’яті, оскільки є обмеження на число циклів запису-очищення одного блоку. З цієї причини для флеш-пам’яті необхідно застосовувати спеціально розроблені файлові системи, які дозволяють рівномірно розподілити цикли запису-очищення по усіх блоках флеш-пам’яті, а так само мінімізувати кількість циклів запису-очищення блоків. Альтернативою є апаратне рішення, в якому підтримується таблиця відповідності між номером логічного блоку і номером фізичного блоку, при

записі даних відповідність між блоками міняється, щоб рівномірно розподілити цикли запису-очищення на усі наявні фізичні блоки.

При проектуванні файлових систем для флеш-пам’яті використовують метод журналювання або ведення історії або логу змін у файловій системі. Записи змін у файловій системі оформляються в один блок, блок очищається і потім записується цілком, оскільки можна провести очищення тільки цілого блоку і різні моделі флеш-пам’яті гарантують різне число гарантованих циклів програмування однієї сторінки. Якщо запис змін у файловій системі перевищує розмір одного блоку, то цей запис розділяється і записується в декілька блоків.

Історія змін у файловій системі повинна дозволити реконструювати стан файлової системи при підключенні флеш-пам’яті. При підключенні (монтуванні) флеш-пам’яті драйвер файлової системи зчитує вміст заголовків усіх записів змін і складає представлення про файли в оперативній пам'яті. При зміні файлової системи запису зі змінами зберігаються у флеш-пам’яті і так само відбиваються в представленні файлової системи в оперативній пам'яті.

Кожен запис про зміну у файловій системі складається з метаданих і відповідних даних. Метадані описують зміну (напр. ідентифікатор файлу, код зміни, зміщення від початку файлу і розмір даних), дані містять самі зміни. Щоб відновити образ файлової системи по історії змін кожен запис для кожного файлу повинен мати номер версії. Неважливо в якому порядку будуть прочитані заголовки змін при монтуванні файлової системи, оскільки по номеру версії завжди можна визначити послідовність їх застосування.

Кожен блок флеш-пам’яті знаходиться в одному з трьох станів. Блок називається «вільним» якщо в ньому не зберігається жоден запис історії змін (такий блок готовий до очищення або вже очищений). Блок називається «чистим» якщо усі записи історії змін записані в ньому містять дійсні дані (розмір даних в такому блоці не можна зменшити). Блок називається «брудним» якщо в ньому є хоча б один запис історії змін, яка повністю або

частково містить недійсні дані (така ситуація виникає якщо подальші зміни для цього ж файлу оновлюють зміни описані в цьому записі). Розмір даних у брудному блоці можна зменшити, оскільки частина даних вже недійсна.

При створенні нової файлової системи уся флеш-пам'ять складається з вільних блоків, які драйвер файлової системи використовує послідовно для збереження історії змін файлів. У якийсь момент часу кількість вільних блоків скоротиться і виникне необхідність створення вільних блоків. Вільні блоки можна отримати за допомогою зчитування декількох брудних блоків, знищення недійсних даних в них, об'єднання записів (сумарний об'єм яких буде меншим ніж початковий об'єм), що залишилися, збереження отриманих записів змін у вільних блоках, при цьому раніше брудні блоки стають вільними.

Процес запису історії змін можна представити таким чином. Один потік записує історію змін послідовно на флеш-пам'ять. Інший потік стежить за кількістю вільних блоків, що залишилися, коли кількість вільних блоків стає дорівнює деякому граничному значенню запускається процес чистки. Процес чистки об'єднує записи у брудних блоках і переміщує їх, а також переміщує чисті блоки. Переміщення чистих блоків потрібне для рівномірного розподілу кількості циклів запису-очищення по усіх блоках флеш-пам'яті. Очевидно, що завжди необхідно мати деяку кількість вільних блоків, тому драйвер файлової системи не повинен дозволити повністю заповнити флеш-пам'ять даними.

Wear-Leveling

Wear-Leveling (урівномірювання зносу) – це технологія продовження терміну служби флеш-пам’яті.

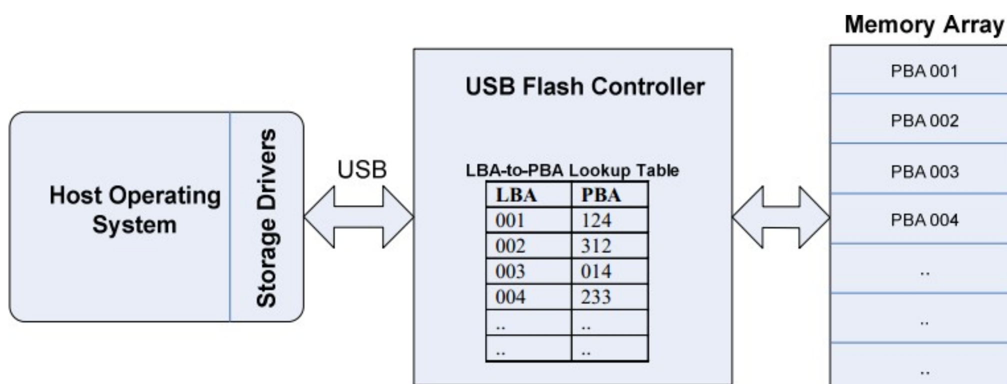
Дані можуть записуватись на флеш-пам’ять кінцеву кількість разів. Не зважаючи на те, що допустима кількість перезаписувань є великою (зазвичай 10 000 або 100 000), якщо запис даних здійснюється на одному і тому ж місці знову і знову, логічно припустити, що пам’ять зношуватиметься в цьому місці. Для уникнення цього здійснюється «урівномірювання зносу» (Wear-Leveling), тобто дані рівномірно розміщуються по всім блокам флеш-пам’яті. Цей процес зменшує сумарний знос пам’яті, тим самим збільшуючи термін її служби.

Щоб зрозуміти як здійснюється Wear-Leveling, слід знати наступні терміни:

LBA (Logical Block Address) – адреса, яка використовується операційною системою для читання або запису блоку даних на флеш-пам’ять.

PBA (Physical Block Address) – фіксована фізична адреса блоку даних на флеш-пам’яті.

Контролер флеш-пам’яті – мікросхема, яка знаходиться разом із флеш-пам’яттю, і предоставляє таблицю відповідностей PBA та LBA (тобто за якою PBA знаходяться дані, що мають відповідний LBA)



Таким чином, таблиця відповідностей PBA та LBA – це щось схоже на зміст книги. Дані можуть бути фізично переміщені, і все, що потрібно буде

зробити - це змінити значення таблиці відповідностей, після чого дані все ще можна буде з легкістю знайти.

Так можна переміщувати дані по всій пам’яті практично без втрат, контролер постійно цим і займається. Обновлені або нові дані записуються у перший-ліпший вільний блок із найменшою кількістю операцій запису. Блок, що містить старі дані стирається у фоновому режимі і позначається як вільний блок. Такий «блокооборот» забезпечує рівномірне зношення блоків пам’яті. Wear-Leveling – це прозорий процес для операційної системи.

Нижче наводиться дуже спрощений приклад.

На початку, у нас є дані за адресою PBA з #1 по #4.

PBA #5 та #6 є "вільними", або порожніми.

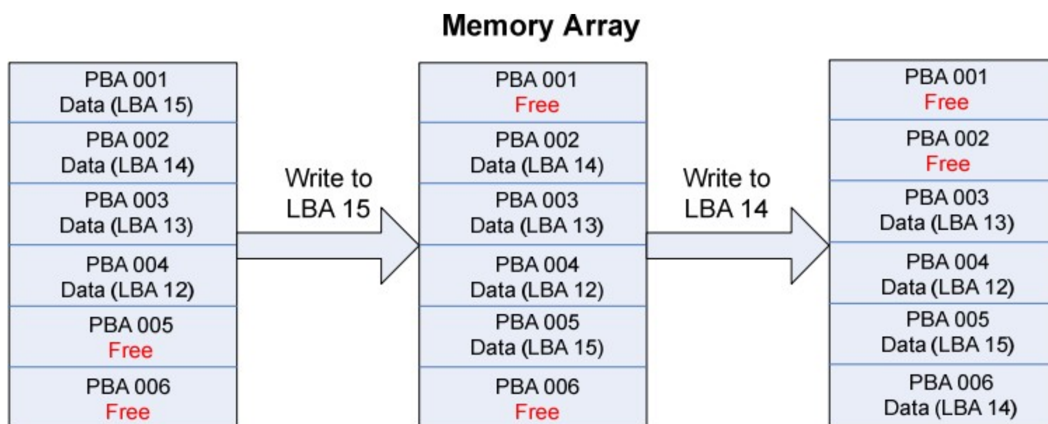


Figure 2

Коли комп'ютер записує нові дані до LBA15, контролер записує їх у PBA #5, а не в PBA #1, якому раніше відповідав LBA15. PBA #1 тепер став порожнім.

Після цього ми будемо записувати LBA14. Як і у випадку із LBA15, дані записуються в новому PBA. Таким чином PBA #2 став вільним.

Допустим, ми записуємо LBA14 ще раз. Знову ж таки, LBA14 тепер матиме нову фізичну адресу.

Статичний та динамічний Wear-Leveling

Як правило, на флеш-пам’яті зберігаються як статичні (записуються лише раз), так і динамічні дані (постійно змінюються і перезаписуються, наприклад, LOG-файли, бази даних і т.д)

Перепризначення статичних даних є більш складною задачею, ніж динамічних даних, оскільки перше вимагає кілька операцій для безпечного руху статичних даних у флеш-пам’яті. Це може сильно вплинути на загальну продуктивність запису.

Wear-Leveling динамічних даних здійснюється за принципом циклічного обороту даних у множині вільних блоків. Динамічний Wear-Leveling сильно зменшує життєвий цикл флеш пам’яті через те, що для обороту даних можуть використовуватись лише вільні динамічні блоки пам’яті.

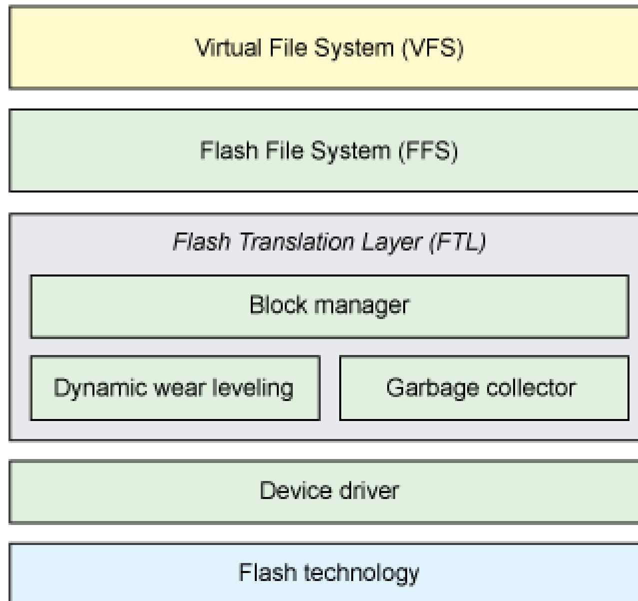
Порівняльна таблиця:

	Статичний	**Динамічний**
Термін служби пам’яті	Дуже довго	Довго
Продуктивність	Повільніше	Швидше
Складність реалізації	Більш складно	Менш складно

Як правило виробники флеш-пам’яті використовують динамічний Wear-Leveling.

Причиною цьому є те, що динамічний Wear-Leveling менш складний у реалізації та забезпечує надійність збереження данних - цього більш ніж достатньо для хорошого попиту на флеш-пам’ять на ринку.

Загальна архітектура файлових систем флеш-носіїв



Як видно з малюнку, нагорі знаходиться шар віртуальної файлової системи (VFS, virtual file system), що надає високорівневий інтерфейс для програм. Нижче йде рівень файлових систем для флеш. Далі розташовується FTL-рівень (Flash Translation Layer, рівень флеш-перетворення), який займається управлінням флеш-носієм, включаючи виділення блоків під дані, перетворення адрес, динамічну оптимізацію зносу і збірку сміття. У деяких пристроях частина функцій FTL-рівня може бути реалізована апаратно.

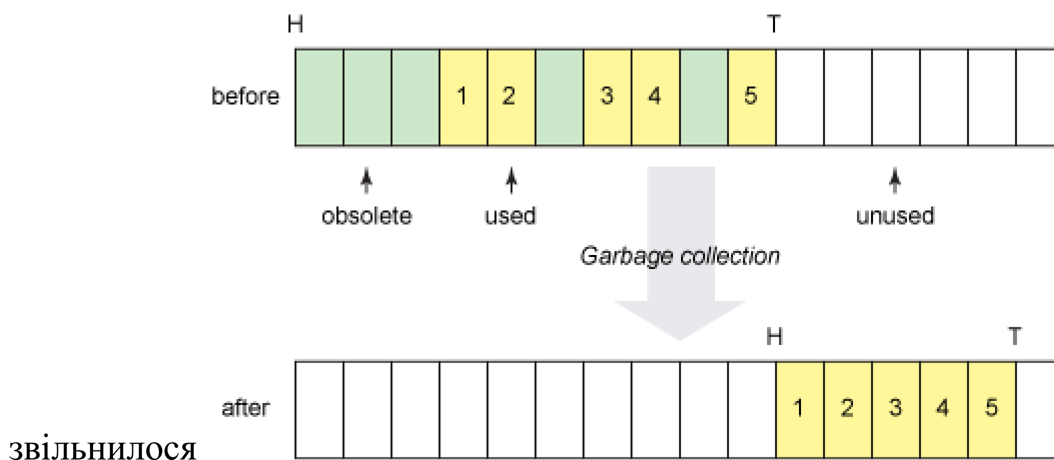
Для взаємодії з флеш-пристроями зазвичай застосовується *інтерфейс пристроїв на основі технологій пам'яті* (Memory Technology Device, MTD). MTD-інтерфейс автоматично розпізнає розрядність шини флеш-пристроїв і визначає необхідну кількість пристроїв для отримання шини необхідної розрядності.

JFFS

Журнальована файлова система для флеш-носіїв (Journaling Flash File System, JFFS) є однією з перших систем для флеш.

JFFS, в основі структури якої лежить журнал, призначалася для NOR-пристроїв. Система на момент створення була унікальною і вирішувала безліч проблем флеш-носіїв, але мала один суттєвий недолік.

JFFS оперує флеш-пам'яттю на основі циклічного журналу блоків. Дані на запис пишуться в блоки з хвоста журналу, а блоки, що знаходяться в голові, готуються для повторного використання. Діапазон між хвостом і головою журналу означає вільне місце.Коли його стає занадто мало, в справу вступає збирач сміття, який знаходить серед застарілих і дефектних блоків блоки з актуальними даними, переміщує їх в хвіст, а потім стирає місце, що звільнилося



В результаті такого підходу відбувається як статична, так і динамічна оптимізація зносу. Проблема даного методу полягає в тому, що відсутня ефективна стратегія стирання - флеш-пам'ять дуже часто перезаписується, в результаті чого знос пристрою відбувається занадто швидко.

Під час монтування структура і розташування блоків зчитуються в пам'ять, тому монтування JFFS-розділу відбувається повільно і використовується досить багато оперативної пам'яті.

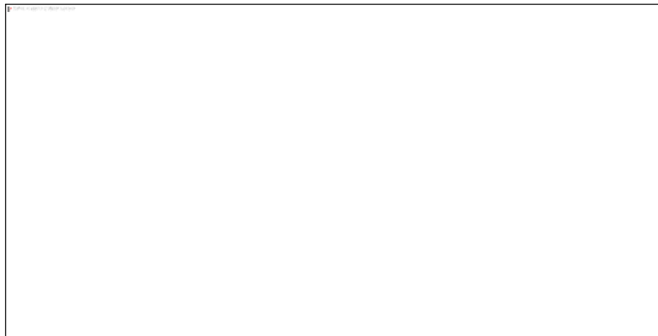
JFFS2

По при всі недоліки, які знижували термін служби FLASH-носіїв відповідно до свого алгоритму оптимізації зносу, система JFFS досить широко застосовувалася. В результаті було вирішено переробити її алгоритм і відмовитися від циклічного журналу. Так з'явилася JFFS2, орієнтована на NAND-пам'ять. Ця система демонструє більш високу швидкість роботи і підтримує функцію стиснення.

JFFS2 розглядає кожен блок флеш-пам'яті незалежно від інших. Для ефективної оптимізації зносу ведуться спеціальні списки. Список чистих блоків відповідає блокам, які містять тільки пакети з актуальними

даними. Список «брудних» блоків містить блоки, кожен з яких має хоча б один пакет із застарілими даними. Список вільних блоків зберігає відомості про блоки, які були стерті і вже готові до використання.

В цих умовах алгоритм збірки сміття може в залежності від ситуації ефективно приймати рішення про те, які блоки готувати до повторного використання. У поточних реалізаціях алгоритм обирає їх зі списків або чистих, або брудних блоків на основі заданої ймовірності. У 99% випадків



використовується список брудних блоків (при цьому актуальні дані переносяться в інший блок). В 1% випадків береться список чистих блоків (при цьому весь вміст просто переноситься в новий блок). У кожному разі цільовий блок стирається і заноситься в список вільних блоків.

Все це дозволяє збирачеві сміття повторно використовувати блоки з неактуальними даними (або частково неактуальними), в той час як дані все одно "проходять" всю пам'ять, що й дає статичну оптимізацію зносу.

Початкові дані

В якості флеш-пам’яті при розробці необхідно використати звичайний файл. Розмір блоку і сторінки у флеш-пам’яті задаються параметрами. Флеш-пам'ять підтримує одноразове програмування сторінки.

Завдання на лабораторну роботу № 8

Розробити драйвер файлової системи для флеш-пам’яті використовуючи метод, описаний в теоретичній частині. У файловій системі є одна

директорія, кількість файлів не обмежена, довжина імені файлу не перевищує 255 символів. Драйвер повинен дозволяти зберігати зміни у файловій системі до тих пір, поки на флеш-пам'яті є вільне місце (тобто не вимагається обробляти брудні блоки).

Розробити консольну програму, що підтримує наступні команди :

- *mount - підключити файлову систему, збережену у файлі;*
- *unmount* - відключити файлову систему, при цьому в пам'яті драйвера не повинно залишитися яких-небудь даних про файлову систему, усі дані мають бути збережені у файлі;
- *blockstat [номерблока]* - вивести інформацію в читабельному вигляді про стан блоку флеш-пам'яті;
- *create ім'я* - створити файл із заданим ім'ям;
- *list* - вивести імена усіх файлів;
- *open ім'я* - відкрити файл із заданим ім'ям, ця команда повинна вивести номер файлового дескриптора *fd*, який можна використати в інших командах;
- *close fd - закрити файл із заданим дескриптором;*
- *read fd зміщення розмір* - прочитати з файлу із заданим дескриптором по цьому зміщенню дані вказаного розміру;
- *write fd зміщення дані - записати у файл із заданим дескриптором по цьому зміщенню дані, зміщення не виходить за межі файлу;*

Звіт

Звіт повинен містити:

7. Опис ідеї реалізації файлової системи.
8. Опис форматів структур файлової системи.
9. Лістинг основної частини розробленої програми.

Завдання на лабораторну роботу № 9

Продовжити роботу над драйвером файлової системи для флеш-пам’яті і додати можливість чистки брудних блоків (тобто реалізувати звільнення місця на флеш-пам’яті за допомогою знищення застарілої інформації у блоках). Також додати можливість створення нових імен для існуючих файлів.

Продовжити роботу над консольною програмою і додати підтримку наступних команд :

unlink ім'я - знищити це посилання на файл (якщо кількість посилань на файл стало рано нулю, то необхідно знищити вміст файлу);

- *link ім'я нове ім’я* - створити нове ім'я для існуючого файлу з вказаним ім'ям.
- *truncate ім'я розмір* - змінити розмір файлу, якщо новий розмір файлу більше ніж старий розмір, то нові дані дорівнюють 0.

Звіт

Звіт повинен містити:

1. Опис алгоритму чистки брудних блоків.
2. Опис форматів структур образу файлової системи в оперативній пам'яті.
3. Лістинг основної частини змін в розробленій програмі.

Електронна версія (мережа КПІ)

ftp://vt513.comsys.ntu-kpi.kiev.ua/pub/edu/spo2/spo2-3,4.doc

Література

5. David Woodhouse JFFS : The Journalling Flash File System.

6. Uresh Vahalia Unix Internals : The New Frontiers.